

NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITET
FAKULTET FOR INFORMASJONSTEKNOLOGI, MATEMATIKK OG
ELEKTROTEKNIKK



HOVEDOPPGAVE

Kandidatens navn: Tanveer Hussain Awan

Fag: TDT4900 Software Engineering, Thesis

Oppgavens tittel (norsk): Eksperimentering i programvareinspeksjoner.

Oppgavens tittel (engelsk): Sources of variations in software inspections: An empirical research project.

Oppgavens tekst:

There is a consensus that code reviews are an effective way to improve software quality and readability. It helps us to improve both our programs and programmers at the same time. However, code reviewing is not a simple task. There are many sources of variations that can affect their outcome, for example reviewer's experience, group size, reading techniques etc. Several empirical studies have been done to identify these sources of variations and to assess how they alter the effectiveness of code reviews. During the last semester, we conducted an empirical study to investigate whether the use of checklists is more effective for smaller inspection teams than the larger ones or vice versa. The main purpose of this project is to design and conduct an empirical study that will help us improve the overall code reviewing process, by finding a good combination of some of these variation factors. The focus will be on investigating if there is any relationship between reviewers' background (education/experience) and the number/types of the defects detected by them.

Oppgaven gitt: 20/01/2004

Besvarelsen leveres innen: 15/06/2004

Besvarelsen levert: 18/05/2004

Utført ved: IDI NTNU

Veileder/Faglærer: Tor Stålhane

Trondheim

Abstract

The major task of this study was to design and conduct an experiment that investigated the effects of the reviewer's background on the number and types of the defects detected by them. During our literature review, we found that there was no definite answer to the effects posed by these sources of variations. Therefore, the question of finding the most effective combination of defect detection technique and the above-mentioned sources of variations became the focus of our experiment.

The subjects included forty-two undergraduate, graduate and PhD students who were supposed to inspect 124 lines of Java code by using the checklist-based method. They were divided into two groups of equal size; the less-experienced subjects and the highly-experienced subjects. Unlike the highly-experienced subjects, the less-experienced subjects had less (or no) experience with industrial software projects and most of them had never participated in a formal software inspection. Most of the highly-experienced subjects were PhD students with industrial experience.

The experiment had one factor with two treatments, since we had two student profiles depending on their experience. All subjects were required to inspect the same client-server Java application. The inspected source was a part of a Java client-server application in which the multi-threaded server encrypted the text given by the client and returned it to the client. The subjects inspected the code individually; there was no group activity involved.

The defect classification used in the experiment was a simple subset of the Orthogonal Defect Classification Scheme [Link6]. We divided the defects into three classes; wrong statements, extra statements, and missing statements. Twelve defects were seeded into the source code and each defect class was represented by four seeded defects. Our hypotheses were that the experienced subjects would be more effective than the less-experienced subjects in detecting defects from all the defect classes, but the results revealed some other patterns.

We suggested a new inspection method that can combine divide-and-conquer and round-robin strategies. These strategies can be used to divide sections of the checklist among the reviewers. This could solve the problem of limited inspection time and ensure that all the sections of the checklist are properly addressed.

We tried to remove the validity threats or at least be aware of the limitations caused by their presence, which in turn helped us to make informed decisions about the best way to reduce their effects. We had limited resources available for the experiment but the ideas that we have gained through this exploration have provided a valuable input for further work in this area.

Preface

This report is the result of the project work done in the course “TDT4900: Software Engineering, Thesis” during the fall semester of 2004. The course was organized by the Department of Computer and Information Science at the Norwegian University of Science and Technology (NTNU). It was part of the 5th year course in the Master of Science degree.

Trondheim, 18 May

Tanveer Hussain Awan

Acknowledgements

I gratefully acknowledge the support of my mentor Professor Tor Stålhane, whose patience and guidance greatly assisted success of this project. Especially, I owe him great thanks for his valuable criticisms and constructive suggestions that essentially helped me organize the experiment during the project. Finally yet importantly, I would like to thank all the students that participated in the experiment.

Index

Abstract.....	1
Preface	2
Acknowledgements	3
Index	4
List of Tables	6
List of Figures.....	7
1. Introduction	9
1.1. Software Inspections.....	9
1.2. Project Context and Motivation.....	12
1.3. Review vs. Walkthrough vs. Inspection	14
1.4. Report Outline	15
2. Pre-study	17
2.1. Types of Inspections.....	18
2.1.1. Fagan’s software inspection	18
2.1.2. FTARM	19
2.1.3. Gilb inspection.....	19
2.1.4. Two-person inspection	20
2.1.5. Active design review	21
2.1.6. Phased inspection	21
2.1.7. N-fold inspection	22
2.1.8. Meeting less inspections.....	22
2.1.9. Code reading.....	23
2.1.10. Code reading by stepwise abstraction	23
2.2. Defect Classification Schemes	24
2.2.1. Orthogonal Defect Classification	24
2.3. Cost and Benefits of Inspections	28
2.4. Summary.....	30
3. Related Work.....	32
3.1.1. Myers 78.....	33
3.1.2. Basili 87.....	34
3.1.3. Porter and Votta 94.....	35
3.1.4. Basili 96.....	35
3.1.5. Porter and Votta 97.....	36
3.1.6. Kelly 97	37
3.1.7. Emam and Wieczorek 98.....	38
3.1.8. Laitenberger 98.....	39
3.1.9. Laitenberger 99.....	40
3.1.10. Basili 2001	40
3.1.11. Awan 2003.....	43
3.2. Summary.....	45
4. Research Method	47
4.1. Types of Research Methods	48
4.2. Experiment Process	53
4.2.1. Experiment definition	53
4.2.2. Experiment planning.....	54
4.2.3. Experiment execution	54

4.2.4. Data Analysis and Interpretation	56
5. Experiment Definition and Planning	58
5.1. Experiment Definition	59
5.2. Subjects.....	62
5.3. Objects	64
5.3.1. Java Source Code.....	64
5.3.2. Defect Classes.....	66
5.3.3. Checklist	67
5.4. Variables	68
5.4.1. Independent Variables	68
5.4.2. Dependent Variables.....	69
5.5. Hypotheses.....	70
5.6. Experiment Design	72
5.7. Threats to Validity	73
5.7.1. Conclusion Validity	73
5.7.2. Internal Validity.....	73
5.7.3. Construct Validity.....	74
5.7.4. External Validity.....	75
6. Experiment Operation.....	77
6.1. Conducted Activities	78
6.1.1. Overview and Training.....	78
6.1.2 Individual defect detection	78
6.1.3. Data Validation.....	80
7. Data Analysis and Interpretation	82
7.1. Data Analysis.....	83
7.2. Data Interpretation.....	91
8. Conclusions and Further Work.....	95
8.1. Summary.....	96
8.2. Conclusions	97
8.3. Recommendations	98
8.4. Further Work	98
8. References	101
Appendix A: Definitions	108
Appendix B: Source code (Without defects).....	109
B.1. EncryptionClient.java	109
B.2. EncryptionServer.java.....	111
B.3. EncryptionServerThread.java	112
Appendix C: Source code (With defects).....	113
C.1. EncryptionClient.java	113
C.2. EncryptionServer.java.....	115
C.3. EncryptionServerThread.java	116
Appendix D: Java Inspection Checklist.....	117
Appendix E: Experience Form	119
Appendix F: Seeded Defects	120
Appendix G: Raw Data.....	121

List of Tables

<i>Table 1 Comparison of review technique features</i>	<i>14</i>
<i>Table 2 Activities and their triggers.....</i>	<i>25</i>
<i>Table 3 Cost of inspections with respect to human effort.....</i>	<i>29</i>
<i>Table 4 Results of inspections with respect to quality.....</i>	<i>29</i>
<i>Table 5 Comparison of inspection techniques.....</i>	<i>30</i>
<i>Table 6 Comparison of research and evaluation</i>	<i>48</i>
<i>Table 7 Qualitative vs. quantitative</i>	<i>51</i>
<i>Table 8 Factors concerning research strategies.....</i>	<i>51</i>
<i>Table 9 Types of experiment contexts.....</i>	<i>61</i>
<i>Table 10 # questions in each section of the checklist.....</i>	<i>67</i>
<i>Table 11 Statistical decision and type of errors.....</i>	<i>71</i>
<i>Table 12 # of defects found by the HE (highly-experienced) and the LE (less-experienced) subjects</i>	<i>85</i>
<i>Table 13 t-Test conducted on the defects data (true defects)</i>	<i>86</i>
<i>Table 14 t-Test conducted on the defects data (missing code statements)</i>	<i>87</i>
<i>Table 15 t-Test conducted on the 'extra code statement' defects.....</i>	<i>89</i>
<i>Table 16 t-Test conducted on the 'wrong code statement' defects.....</i>	<i>90</i>

List of Figures

<i>Figure 1 Phases in Fagan's inspection process</i>	18
<i>Figure 2 Phases in FTARM's inspection process</i>	19
<i>Figure 3 Phases in Gilb's inspection process</i>	20
<i>Figure 4 Phases in ADR</i>	21
<i>Figure 5 Phases in single and multiple-phases inspection</i>	21
<i>Figure 6 Phases in N-Fold inspection</i>	22
<i>Figure 7 Levels in ODC</i>	24
<i>Figure 8 Target entities in ODC</i>	26
<i>Figure 9 Defect types in ODC</i>	26
<i>Figure 10 Qualifiers in ODC</i>	27
<i>Figure 11 Defect age in ODC</i>	27
<i>Figure 12 Defect sources in ODC</i>	27
<i>Figure 13 Cost of rework in software development</i>	28
<i>Figure 14 Percentage of rework in software development</i>	28
<i>Figure 15 Three dimensions of the research methods</i>	49
<i>Figure 16 Types of research purposes</i>	49
<i>Figure 17 Types of research processes</i>	50
<i>Figure 18 Types of research outcomes</i>	51
<i>Figure 19 Phases of experimentation and their input/output</i>	53
<i>Figure 20 Experiment planning phase</i>	54
<i>Figure 21 Experiment execution phase</i>	55
<i>Figure 22 Analysis of the collected data</i>	56
<i>Figure 23 GQM Planning Phase</i>	59
<i>Figure 24 Individual review activity and the factors that affect its effectiveness</i>	79
<i>Figure 25 compares the number of subjects in each group which detected the given defects (D=Defect)</i>	83
<i>Figure 26 Comparison of effectiveness in defect detection (Mean values)</i>	84
<i>Figure 27 'Missing statement' defects detected by the subjects</i>	87
<i>Figure 28 'Extra statement' defects detected by the subjects</i>	88
<i>Figure 29 'Wrong statement' defects detected by the subjects</i>	89
<i>Figure 30 # subjects who successfully detected the defects</i>	90

1. Introduction

*Intellectuals solve problems; geniuses prevent them –
Albert Einstein*

1.1. Software Inspections

Software inspection is a static method to verify and validate a software artifact manually [Fagan76]. Verification involves checking that the product is developed correctly, i.e. conforms to its specification. Validation involves checking that the correct product is developed, i.e. meets the expectations of the customer. Michael Fagan is the pioneer of the inspection process [Fagan76]. He introduced the inspection process in 1976 during his work at IBM. Software inspections are not limited to source code; any artifact produced during software development can be inspected.

The IEEE defines software inspection as [IEEE91]:

‘... a formal evaluation technique in which software requirements, examined in detail by a person or group other than the author to detect development standards, and other problems...’

The validation and verification process involves both dynamic and static techniques to check and analyze the system. Static techniques are used to check and analyze the system representations such as requirement document, design document, and source code. Static techniques can be applied at all stages of the software development process. Dynamic techniques or tests can only be used when an executable program is available [Sommerville92]. Static techniques cannot completely replace dynamic techniques in the verification and validation process because each technique detects different types of defects with different efficiencies. Static techniques can only check the correspondence between a program and its specification but these techniques cannot demonstrate that the software is operationally useful [Sommerville92]. It cannot predict the dynamic behavior of the program. Static and dynamics techniques are complementary and both need to be used in the validation and verification process.

Inspections can have a significant cost saving affect because defects are found early in the life cycle and less effort will be spent on debugging. The defect correction cost increases significantly for each successive stage and defects from the earlier stages have greater rework costs the later in the development cycle that they are discovered. It is a good way to enforce standards and provide necessary information for defect prevention activities such as root cause analysis. Several studies have shown that the costs for defect detection and defect correction during inspections are much lower than other techniques, such as testing. According to Laitenberger et al, since inspection is a human-based activity, its costs are determined by human effort involved in inspection process [Laitenberger02].

Inspection process consists of three main steps: defect detection, defect collection, and defect correction. Each of these steps has several phases. The first stage is planning. In this stage, it is the moderator's responsibility to select the participants and prepare the required material. During the next stage, the inspection team gets an overview of the document under inspection. This stage is not necessary if all of the team members are already familiar with the document to-be-inspected. The third stage involves individual preparation of each team member. Each team member examines the inspected documents and notes the defects found. During the fourth stage, an inspection meeting is held. The entire team attends this meeting. The main purpose of this meeting is to record and categorize the defects, so the author can fix them later. This meeting should not be longer than two hours. The fifth stage involves reworking, where the author corrects the identified defects. In the last stage, the moderator considers if a re-inspection of the document is required. If all the identified defects are addressed properly then no re-inspection is needed.

IEEE recommends five roles for the team members conducting the inspection; moderator, author, reader, recorder, inspector. The moderator is responsible for leading the inspection. Ideally, the moderator should be a person that does not have a stake in the project. His job is to ensure that the inspection is going according to the plan. He has the responsibility for scheduling and controlling the meetings, reporting inspection results and following up on rework issues. The moderator should have formal training in how to conduct inspections. The author is the creator or maintainer of the product under inspection. The author attends the group meeting and assists by answering questions asked about the product. The author cannot take the role of moderator, reader or recorder. The author is responsible for doing the rework necessary to resolve the issues rose during the inspection. The reader is responsible for reading through the product and explaining the meaning of it to the inspectors. Normally, the reader paraphrases what is happening in the product, but he does not read the product verbatim. The recorder is responsible for classifying and recording the defects and issues, raised during the inspection. The moderator can perform the role of the recorder. The inspectors are responsible for identifying the anomalies in the product. It is a good idea to include representatives for all the stakeholders in the inspection team.

The following list of pre-condition given by [IEEE97] must be met before starting an inspection:

- A statement of objectives for the inspection
- The software product to be inspected
- Documented inspection procedure
- Inspection reporting forms
- Current anomalies or issues list
- Inspection checklists
- Any regulations, standards, guidelines, plans, and procedures against which the software product is to be inspected
- Hardware product specifications
- Hardware performance data

Many organizations have benefited from inspections; however, several factors can undermine the effectiveness of inspections. By avoiding these pitfalls, we can reap the benefits of inspections.

The major challenge of effective inspections is having the participants prepare thoroughly for the inspection meeting. The main problem is that the participants are not given sufficient time for individual preparation. Since a large percentage of the detected defects are found during individual preparation, this pitfall can badly affect the effectiveness of inspections. The moderator should reschedule the meeting, if he finds out that the participants are not adequately prepared. Inspections should inspect the product, not the producer. Criticism should be done in such a way that the author does not feel that his professionalism is being questioned. If issues are raised in a confrontational style, the author will be reluctant to submit his future products for inspections.

Inspections require significant time and resources and it is therefore important to keep them focused on the defect detection. The participants should not waste their valuable time on discussing possible resolutions to detected defects and issues. The moderator should control the meeting by bringing the discussion back on track, to focus on defect detection, rather than defect correction.

Selection of a realistic review rate is an important issue. If the review rate is too high, many errors will remain undetected because inspectors would not have enough time to focus on each part of the document. On the other hand, a too low review rate can make the cost of detecting each error significantly higher. Fagan [Fagan76] has suggested reviewing about 500 LOC per hour during the overview stage, about 125 LOC per hour during individual preparation stage, and 90 to 125 LOC per hour during inspection. He suggests that the maximum time spent on inspection should not exceed two hours.

Participant should have common understanding of their roles and responsibilities. A lack of common understanding about inspection process can lead to inconsistencies in inspection's objectives and approaches. The participants cannot effectively contribute in defect detection, if they do not have appropriate skills for it. Ideally, the participants should have experience in creating the type of the artifact that is under inspection. Otherwise, inspection become more of an exercise in training and is not likely to improve the quality of the product

1.2. Project Context and Motivation

We will focus on two factors of code reviewing process; reviewer's experience and classes of defects detected by them. The goal is to answer the following questions:

How important is the reviewers' experience in determining their effectiveness in defect detection during the individual review?

What is the relationship between the reviewers' experience and classes of defects detected by them?

What classes of defects are normally detected/missed by the experienced reviewers?

Are there any types of defects that are generally detected more easily by less experienced reviewers than highly experienced reviewers?

Is there any uncontrolled variable that affects these two factors or interaction between them?

If the classes of detected defects are the same regardless of the reviewer's experience; is it worth that the experienced developers spend their time on code reviewing? If the classes of detected defects differs according to the reviewer's experience; is it not more effective (in terms of cost and number of detected defects) to have two less-experienced reviewers than a single experienced reviewer. Is there any defect-type and experience related criteria that can guide a project leader to form an ideal reviewing team with members that have different experience and background? Can such approach ensure that all classes of defects are detected?

The results from [Myer78] indicated that the inspection methods were more predictable than computer-based testing methods and the number and the classes of the defects detected by experienced subjects varied significantly. They found that most of the reviewers focused on the defect classes related to the program's logic.

The experiment conducted by [Basili87] showed that the fault detection rate, fault-detection-effort, and the number of detected faults were dependent on the application domain of the program. They found that the code reviews were effective in finding interface faults but they could not correlate this to the reviewer's experience.

Porter and Votta [PorterVotta94] compared scenario-based reading with checklist-based reading and adhoc reading. They classified the defects into omissions and commissions. The results showed that the scenario-based reading was more effective in finding the defect classes the scenario was designed to uncover, regardless of the reviewer's experience.

In another experiment [PorterVotta97], they found that different reviewers found different number and types of defects because they were looking for different kinds of issues e.g. the authors found the defects related to software maintenance issues since they had a higher concern for its maintainability than the rest of the reviewers.

In [Laitenberger96], the subjects were classified into three perspectives; designers, testers, and users, according to their experiences and tried to find if the reviewer's experience in the role influenced their effectiveness. They did not find any statistically significant results.

The experiment conducted by [Kelly97] showed a great variation between the subjects' effectiveness, depending on their experience. The results showed that the experienced inspectors identified a higher proportion of findings that required deeper understanding of the code. They did not provide any statistically significant result. They claimed that inspectors require a minimum level of experience to be effective.

The experiment conducted by [Basili01] investigated the effect of reviewer's technical experience, domain knowledge, and software development experience on their defect detection effectiveness and efficiency. The study did not investigate the relationship between the reviewer's experience and the classes of defects detected by them. Their statistically significant results showed that the reviewers' domain knowledge does not improve their effectiveness but it made them more efficient in defect detection. The reviewers' technical experience did not show any affect on their effectiveness and efficiency. The reviewers' software development experience did not improve their effectiveness; on the contrary, the number of defects found by the low experienced reviewers was higher than that of the high experienced reviewers. The experienced reviewers were less efficient than the less experienced reviewers.

All the above-mentioned experiments are discussed in the next chapter. The discussion has shown that conducting effective code inspections is not a trivial task. Despite of the extensive research conducted in the field of software inspections, there are still many controversies regarding the reviewer's experience and the number and the types of the defects detected by them. It is in this light, the relation between these two variables becomes interesting for us.

1.3. Review vs. Walkthrough vs. Inspection

Use walkthroughs for training, design reviews for consensus, but use inspections to improve the quality of the document and its process [Gilb93].

Walkthrough is a relatively informal review technique in which the author leads the reviewers through a document and the reviewers identifies possible problems and improvements [Link2]. The main purpose is to find faults; their correction is not considered. Walkthrough does not involve use of checklists. The document under inspection is not gone through line by line but in a process-oriented way. The role of the author is the most important one in a walkthrough. The author can influence the assessment up to a certain level by directing the focus of his presentation to less critical parts of the document [Link3].

The inspection process is more formal than the walkthrough. The planning phase requires more resources than the planning phase in a walkthrough. The inspection process is longer than the walkthrough; however, the extra time and cost is justified by their efficiency in defect detection in the early phases of the development process when they are easiest and least costly to correct [Acerman83].

Design review is a formal documented and systematic study of a design proposal done by experts. The experts responsible for design reviews are not necessarily engaged in the design [Gilb93]. In contrast to inspections and walkthroughs, the focus of reviews is not on detecting technical flaws, but on ensuring that the design meets the project requirements and provides customer satisfaction. Reviews are effective early on during requirements verification and conceptual model validation [Hollocker87]. Peer review is the least informal technique carried out by colleagues.

The following table compare features of review techniques [Gilb93]:

	Inspection	Walkthrough	Design Review	Peer Review
Proven cost-effectiveness on all types of documents	Yes	No	No	No
Metrics feedback into process	Yes	No	No	No
Trained leader	Yes	No	Yes	No
Uses sources	Yes	No	Yes	No
Uses standards	Yes	Yes	Yes	No
Meeting control	Yes	No	Yes	No
Entry and exit control	Yes	No	Exit (Yes)	No
Authorship standards improvement due to learning	Yes	No	No	No
Use of experts and peers	No	Yes	Yes	Yes

Table 1 Comparison of review technique features

1.4. Report Outline

The rest of this report is divided into the following six chapters:

Title	Description
Pre-study	This chapter includes background theory about software inspections and discusses the costs and benefits of conducting inspections. It presents results from several industrial case studies and research projects.
Related work	This chapter describes existing work that has relevance to this study.
Research methods	This chapter introduces several types of research methods and explains how this study uses a combination of them. It also classifies the research methods according to their purpose, process and outcome. The main phases of the experiment process followed in this study are also explained in this section.
Experiment definition and planning	This chapter provides the formal definition for the experiment. It explains the context and design of the experiment. First, it gives a full description of the subjects and objects involved in the experiment. Then it explains the dependent and independent variables, and how they will be measured and analyzed. The hypotheses and the strategies to test them are also stated in the chapter. At the end, it discusses the major validity threats and how they are handled in the experiment design.
Experiment operation	This chapter presents how the activities were conducted during the experiment operation. It also describes the work done for subjects' preparation and how the data was collected during the experiment.
Data analysis and interpretation	This chapter is a presentation of the raw data and their analysis. It describes the statistical models and calculations used in the data analysis. To give a better insight into the prerequisites for the data analysis, it explains information like sample sizes, significance levels and application of tests. It also explains the rejection of the hypothesis or the inability to reject the null hypothesis. We discuss the factors that could have affected our results. In this way, it considers the validity of the experiment to avoid making false conclusions.
Conclusions and further research	It contains our conclusions and suggestions for the future research.

2. Pre-study

In ancient China, there was a family of healers, one of whom was known throughout the land and employed as a physician to a great lord. The physician was asked which of his family was the most skillful healer. He replied:

- *"I tend to the sick and dying with drastic and dramatic treatments, and on occasion someone is cured and my name gets out among the lords."*
- *"My elder brother cures sickness when it just begins to take root, and his skills are known among the local peasants and neighbors."*
- *"My eldest brother is able to sense the spirit of sickness and eradicate it before it takes form. His name is unknown outside our home."[\[Link1\]](#)*

This chapter provides an overview of state-of-the-art within the area of software inspections. It includes the following sub-sections:

- **Section 2.1 ‘Types of inspections’:** This section introduces and compares the most important inspection techniques. For each of these techniques, it gives a short description of the necessary inputs and outputs of the inspection process
- **Section 2.2 ‘Defect Classification Schemes’:** This section introduces Orthogonal Defect Classification scheme and also discusses some general theory about the defect classification schemes.
- **Section 2.3 ‘Costs and benefits of inspections’:** This section discusses the costs and benefits of conducting inspections. It presents results from several industrial case studies and research projects that have investigated the cost-effectiveness of inspections.
- **Section 2.4 ‘Summary’:** This section provides a brief summary of chapter 2.

2.1. Types of Inspections

This section gives a short description of the most important inspection techniques.

2.1.1. Fagan's software inspection

Michael Fagan defined the original inspection process in 1976 [FAG76], with the main purpose of detecting and removing defects as early as possible in the software development cycle. He realized that the success of a project depends heavily on managing its internal processes. He emphasized on the use of detailed exit criteria as process control checkpoints.

A typical Fagan's inspection involves four to six participants, with each participant having a well-defined role in the inspection. It includes author, moderator, reviewer, reader and recorder. Fagan's inspection process is rigorously structured and meeting-oriented [IEEE97]. It has three phases; organization, detection, and completion.

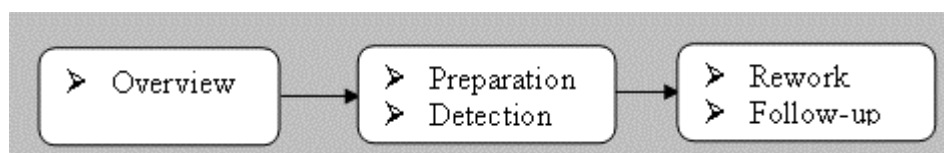


Figure 1 Phases in Fagan's inspection process

The organization phase involves an overview stage, which is an overview meeting that involves the entire team. The main activities include, defining the objectives of the inspection, assignment of roles and distribution of inspection material.

The detection phase involves two stages; preparation and detection. In the preparation stage, each team member carries out individual preparation by inspecting the document and recording the potential defects. The use of checklists can help inspectors focus on certain parts of the document. In the detection stage, the moderator leads this meeting. The entire team meets and collaboratively identifies potential defects. Defect detection should be the sole purpose of this meeting; discussions concerning causes or solutions of the defects should be avoided since they might degrade the inspection results. The meeting should be rescheduled, if the moderator notices that the participants are not prepared. This meeting should not be longer than two hours. The moderator writes a report that includes the inspection details and the detected defects. This report is submitted to the author.

The completion phase includes the rework stage and the follow-up stage. During the rework stage, the author carries out the necessary modifications to handle the defects and issues found during the inspection. In the last stage, the moderator ensures that all the required modifications have been made. The moderator then decides whether a partial or full re-inspection of the document is required.

2.1.2. FTARM

Philip Johnson developed an asynchronous inspection method called Formal Technical Asynchronous Review Method (FTARM), which removes the need to have face-to-face group meetings [Johnson93] [Johnson94]. FTARM is a computer-based method, which provides an asynchronous discussion environment. It allows the users to access an online version of the document under inspection. Reviewers can add their comments to the document that will be visible to other reviewers.

A review tool called Collaborative Software Review System (CSRS) has implemented FTARM [Johnson93]. As with conventional inspection, FTARM defines several roles: moderator, producer and reviewer. FTARM process consists three phases and seven stages.

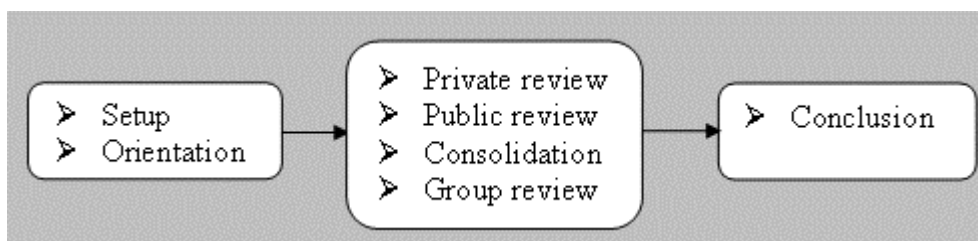


Figure 2 Phases in FTARM's inspection process

The organization phase involves setup and orientation stage. The setup stage selects the inspection team and prepares the document for inspection via CSRS. The document under inspection must be split into a set of hypertext nodes. The orientation stage is equivalent to the overview stage in Fagan's inspection and involves preparation of the participants. The private review stage is same as preparation stage in Fagan's inspection. Each participant examines the hypertext nodes of the document. Participants can make new nodes containing new issues and comments. Comment nodes are visible to the entire team, which allow inspectors to obtain guidance from each other. Issue and action nodes do not allow public access.

The detection phase involves four stages. In the public review stage, all the nodes created by reviewers allow public access to the entire team. Reviewers can asynchronously vote for each comment. Vote cast can be one of three; confirm the issue, disconfirm the issue, or neutral. This stage ends when all the nodes have been resolved or if the moderator decides that further discussion would not be useful. During the consolidation stage, the moderator summarizes the results of the inspection into a report and submits it to the document's author. The report should include reviewers' opinion about resolved and unresolved issues, and their degree of consensus. In the group review meeting stage, the moderator can decide to arrange a face-to-face group meeting, if there are still any unresolved issues. During the completion phase, the moderator produces the final inspection report and a metrics report.

2.1.3. Gilb inspection

The phases in Gilb's inspection are almost similar to Fagan's inspection except the brainstorming phase right after the inspection meeting.

The organization phase is divided into three stages; entry, planning, and kick-off. The entry stage is started when certain entry criteria have been satisfied. The planning stage involves selection of the team members and making an inspection plan. The Kick-off stage gives an overview of the documents and the inspection plan to all of the participants. The normal duration for this stage is 30 minutes.

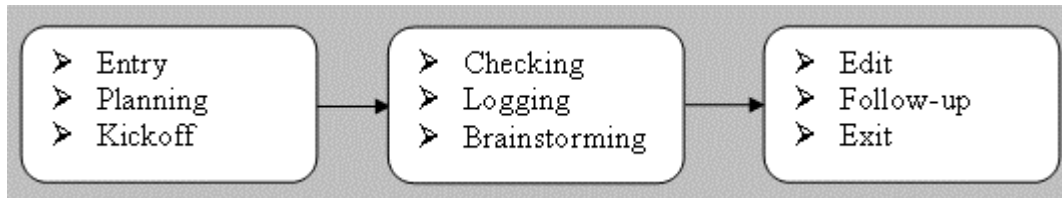


Figure 3 Phases in Gilb's inspection process

In the detection phase, the stages of checking, logging, and brainstorming are conducted. During the checking stage, each participant checks the documents for defects. They should also note the points that need improvements, to make the current work process more efficient. In the logging stage, the entire team meets and collectively logs all the detected defects. The leader is responsible for leading this meeting. Discussions should be avoided at the logging meeting because it takes up all team members' time and steals attention from defect detection. This meeting should not be longer than 2 hours. T

The optional stage of brainstorming involves causal root cause analysis meeting. The main purpose of this activity is to improve the development processes by preventing causes of discovered defects from happening in the future. The recommendations to improve the inspection procedure and standards are also handled in this activity. The maximum duration for this meeting is 30 minutes.

The completion phase involves three stages; edit, follow-up, and exit. During the edit stage, the author receives the defect log and makes necessary changes in the document. In the follow-up stage, the inspection leader ensures the correctness of the modified document. He should also check that edit process has been properly followed. The exit stage is an objective form of approval. The main purpose of this activity is to check if the product is economic to release. The inspection leader should follow the determined exit criteria and not exit the inspection, unless exit criteria are properly met.

2.1.4. Two-person inspection

David B. Bisant [Bisant89] proposed that a two-person inspection method could be a useful approach when faced by tight resource constraints. They recommend including only the author and one reviewer in the inspection team, and removing the moderator role.

The process consists of a single session and uses only ad-hoc techniques during the preparation. It is a less costly approach than the conventional larger team methods and is more effective in improving the performance of the slower programmers [Bisant89].

2.1.5. Active design review

Active Design Review is a defect detection method proposed by Parnas and Weiss [Parnas85]. ADR ensures thorough coverage of design documents by associating specific responsibilities with each reviewer concentrating on different types of errors. The main reason for selecting this approach was that in conventional design inspections, inspectors are required to examine too much information, which significantly affects the useful interaction between inspectors and author.



Figure 4 Phases in ADR

The organization phase involves an overview stage which includes activities like preparation of the design and documentation for review, identifying the specialized review, selection of the reviewers according to their specialties and designing of questionnaires.

The detection phase has a review stage where each reviewer acts proactively on his part of the artifact by using the given questionnaire and conducts the review as several smaller and specialized sessions.

During the completion phase, individual meetings between the author and each reviewer are conducted to collect feedback. One-on-one meeting with the author makes it easier for reviewers to speak up. The advantage of ADR is that the reviewers focus on the areas they are best suited to evaluate and makes effective use of reviewer's time.

2.1.6. Phased inspection

Knight and Myers introduced this method in 1993 [Knight93]. Phased inspection addresses both errors and internal properties of the software such as maintainability, reusability and portability.

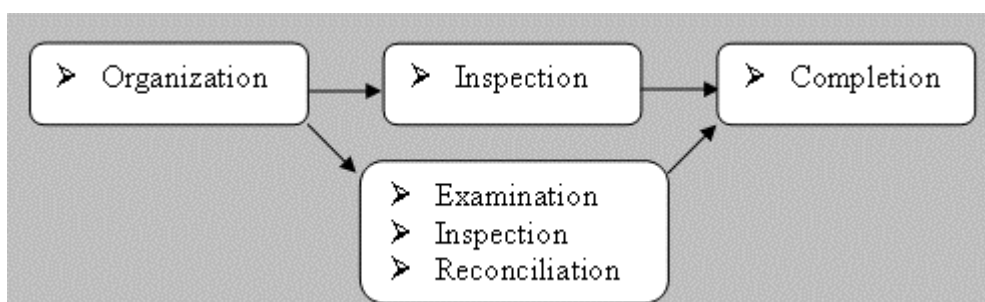


Figure 5 Phases in single and multiple-phases inspection

This method divides each inspection into several mini phases. In each mini phase, one or more inspectors can participate to detect defects of one particular type. Unlike ADR, in phased inspection, the execution of mini phases is serial. When the inspection phase involves more than one inspector (multiple-inspector mode), there is an additional step called

reconciliation. In the reconciliation step, all inspectors meet to collate their defect lists. Unlike ADR, phased inspection also checks the internal properties of the software under inspection.

2.1.7. N-fold inspection

G. M. Schneider et al [Schneider92] introduced this method. They based their study on the hypothesis that a single inspection team can detect only a subset of the defects present in the inspected artifact, while multiple groups can detect more unduplicated defects.

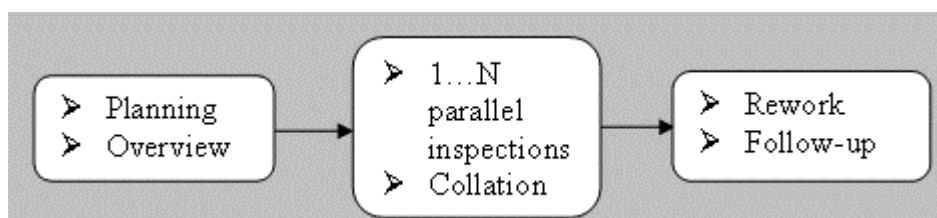


Figure 6 Phases in N-Fold inspection

N-fold inspections divide the inspecting team into sub-groups. Each of these sub-groups carries out independent defect detection of the same artifact at the same time. Each sub-group holds its own team meeting at the end of each session. The moderator for all of the sub-groups will be the same person. His job is to control the overall process, which includes gathering the data from all the sub-groups and removing the duplicates from the defect lists.

2.1.8. Meeting less inspections

Meetings are one of the most discussed activities in the inspection process. The experiments conducted by Land [Land97] and Porter [Porter97] found little synergy in inspection meetings. Furthermore, delays due to inspection meeting scheduling can significantly lengthen development time, which can add ~ 20% to the project costs [Ballman94] [Russel91]. Votta states that meeting costs outweigh their benefits, and proposes use of depositions instead of meetings. In depositions, the author (and possibly the moderator) can meet separately with each reviewer [Votta93].

Fagan has recommended keeping the inspection team to four persons [Fagan76]. Bisant and Lyle have found performance advantages in an experiment with two persons: one inspector and the author, where the author can also act as an inspector [Bisant89]. The studies conducted by Madachy showed that the optimal size is between three and five people [Madachy93]. Votta et al found that reducing the number of inspectors from two to four persons may significantly reduce effort without increasing inspection interval or reducing effectiveness [Votta97]. Tripp et al found empirical evidence confirming the hypothesis that N different teams will detect more defects than a single large inspection team [Tripp91].

Parnas [Parnas85] made a suggestion to keep the number of team members attending the collection meetings to a minimum. According to Parnas, when n team members are attending a collection meeting, only two of them will interact in a productive way, while other team members (n - 2) will only be listening or not doing anything useful at all. Fagan states that the group meetings provide a synergy effect. However, the experiments conducted by Land [Land97] and Porter [Porter97] found no significant synergy effect.

Several small teams can be less effective and more expensive than one large team. On the other hand, a small team can be more flexible than a large team. With a large team, it is harder and more time consuming to conduct a meeting, however, more participants can generate more ideas and subsequently find more defects together. It is obvious from the above discussion that there is no definite answer to the optimal number of participants in software inspections.

2.1.9. Code reading

Steve McConnell proposed code reading as an alternative to formal code inspections [McConnell93]. It is an informal peer-review process involving a small group of participants. Reading through the source code is the only main activity of this technique. The optimal rate of code reading is 1K lines per day. An optional meeting can be arranged to discuss the defects with the author. Code reading emphasizes more on individual defect detection than group activity.

2.1.10. Code reading by stepwise abstraction

H. D. Mills [Mills79] proposed the idea of reading by stepwise abstraction. The first step is to divide a code document into a set of subprograms. This decomposition continues until the point when a subprogram cannot be further decomposed (prime subprogram). The next step is to determine the function of each prime subprogram. This information is later combined to determine the function for the entire program. The last step is to compare the program's derived function with the original specification of the program. The study done by V. R. Basili et al. [Basili87] confirms that this technique is more efficient and effective in detecting defects than functional or structural testing.

2.2. Defect Classification Schemes

Defects can be classified by analyzing and categorizing software defects. It is not a trivial task. There are many possible ways of categorizing the defects, but if the classification is not done properly, there can be a number of shortcomings involved with this process e.g. creating too many or too few categories, ambiguity in different classes etc. Researchers have proposed various defect classification schemes. Most of these schemes revolve around one or more of the following three points:

- The injection phase when the defect was produced e.g. design or coding phase of software life cycle.
- The defect type i.e. the structure of the defect.
- The defect reason i.e. what sort of error is responsible for the defect

The root cause for any defect can be either local or external. The external root causes are the ones that are not in developer's control e.g. an external API that generated an error.

2.2.1. Orthogonal Defect Classification

ODC was developed by IBM [Link6]. The levels in ODC are shown in the following figure.

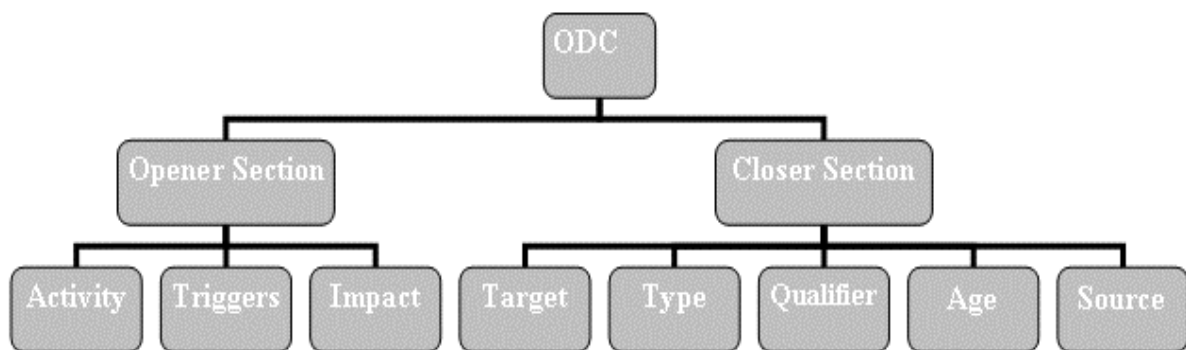


Figure 7 Levels in ODC

ODC has divided the defect attributes into two main sections; opener section and closer section. The attributes in the opener section are available when the defect is discovered. The attributes in the closer section becomes available after the defect is fixed.

Activity: These defect removal activities refer to the activity that helped to detect the defect. ODC divides these activities into design reviews, code inspections, unit testing, functional testing, and system testing. The design reviews are used to either review a new design or compare an already existing design with the given requirements. The code inspections are used to ensure that the code properly implements the documented design. The unit testing involves a variety of white box testing techniques. The functional testing executes the program to check if the software fulfills its functional requirements. During the system testing, the complete system is executed.

Triggers: Triggers are the conditions that are needed to reproduce the defect. ODC classifies the trigger according to the defect removal activities.

Defect Removal Activity	Triggers
Design review	<ul style="list-style-type: none"> • Design Conformance • Logic/ Flow • Backward Compatibility • Lateral Compatibility • Concurrency • Internal Document • Language Dependency • Side Effect • Rare Situations
Code inspections	<ul style="list-style-type: none"> • Design Conformance • Logic/ Flow • Backward Compatibility • Internal Document • Lateral Compatibility • Concurrency • Language Dependency • Side Effect • Rare Situations
Unit testing	<ul style="list-style-type: none"> • Simple path • Complex path
Functional testing	<ul style="list-style-type: none"> • Coverage • Variation • Sequencing • Interaction
System testing	<ul style="list-style-type: none"> • Workload/Stress • Recovery/Exception • Startup/Restart • Hardware Configuration • Software Configuration • Blocked Test

Table 2 Activities and their triggers

Impact: The impacts that a customer would have faced, if the defect had not been detected. ODC has defined the following impacts:

- **Installability:** Can the customer install and use the software?
- **Serviceability:** Can we quickly diagnose the failures to help the customers?
- **Standards:** Does the software uses the current standards?
- **Integrity/Security:** Is the system integrity and security good enough?
- **Migration:** Can the system be easily upgraded to a new version?
- **Reliability:** How reliable is the system?
- **Performance:** What is the speed of the system in performing the given tasks?

- **Documentation:** Does the system provide useful documentation that explains the intended use of the system?
- **Requirements:** What user requirements have not been fulfilled by the current version of the system?
- **Maintenance:** Is it easy to apply preventive and corrective maintenance tasks?
- **Usability:** Can the system be easily used by the users?
- **Accessibility:** Can the users with physical disabilities use the system?
- **Capability:** Does the current version of the system fulfill the known requirements and perform the intended functions?

Defect target: The entities that were fixed by discovering the defect. The figure given below shows different types of targets used in ODC:

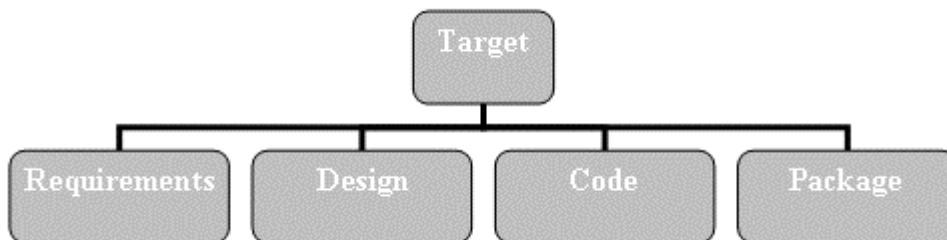


Figure 8 Target entities in ODC

Defect type: Refers to the correction that was made to correct the defect. ODC divides the defect types into many classes as shown in the figure given below:

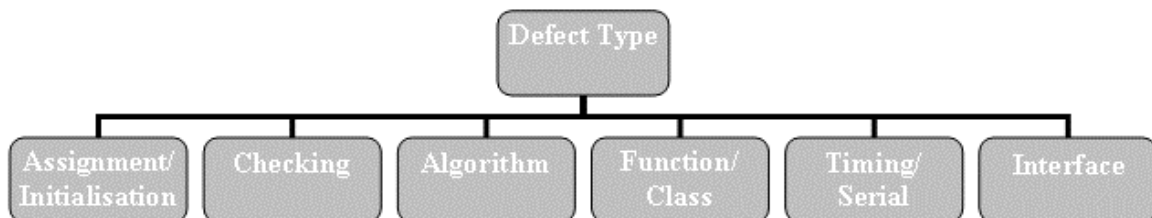


Figure 9 Defect types in ODC

The assignment and initialization defect type refers to the situations when a variable value is not assigned or initialized properly. The checking defect type is related to validation of parameters. The algorithms defects type occurs when an algorithm is incorrectly implemented. The function and class defect types normally require changes in the design. Wrong serialization of the system's resources can cause the timing and serial defect types. The interface defect types refer to communication errors between modules, components, drivers, objects and functions. There can also be relations defect types due to associations among functions, objects and data structures

Defect qualifier: Defect qualifiers are applied to defect types and refers to wrong implementation of an element. There are three types of qualifiers, as shown in the figure given below:

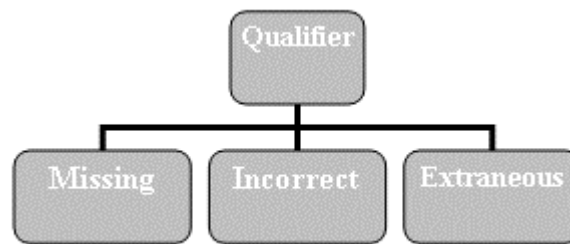


Figure 10 Qualifiers in ODC

Missing qualifiers are basically caused by omission defects, while incorrect qualifiers are due to commission defects. Extraneous qualifiers are due to the defects that are caused by some external code or document.

Defect age: This is related to the history of the target in which the defect is discovered. Refix defects are the ones that are caused during the correction of some other defect. Rewritten defects are caused when a function or design document is rewritten. New defects are caused by the implementation of new functions in the current version of the system. On the other hand, base defects are the ones that are not caused by the current version of the system.

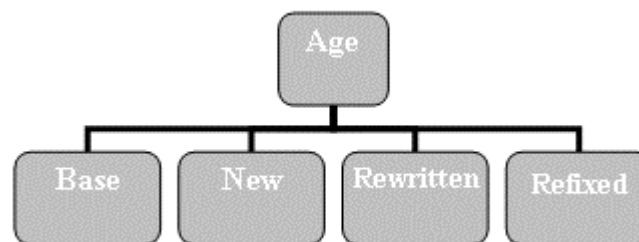


Figure 11 Defect age in ODC

Defect source: This refers to the source document that was origin of the defect. In-house refers to the documents that are developed by the organization's own team. Reuse library sources are the ones that are caused by the use of a standard reuse library. The defects which originated from an outsourced work are classified as outsourced. Ported defects happen when a system is used in a new platform.

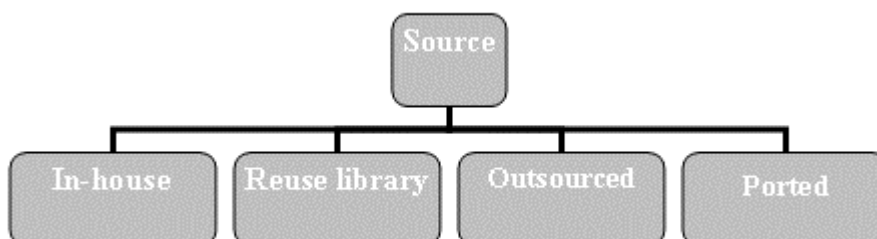


Figure 12 Defect sources in ODC

2.3. Cost and Benefits of Inspections

Figure 14 shows that the defect correction cost increases significantly for each successive stage and defects from the earlier stages have greater rework costs the later in the development cycle that they are discovered. As shown in Figure 13, 44% of typical software development effort is devoted to defect-correction (rework). The use of inspections can significantly reduce the amount of rework needed because defects are found early in the life cycle. Both figures are derived from the data published by Boehm [Boehm87].

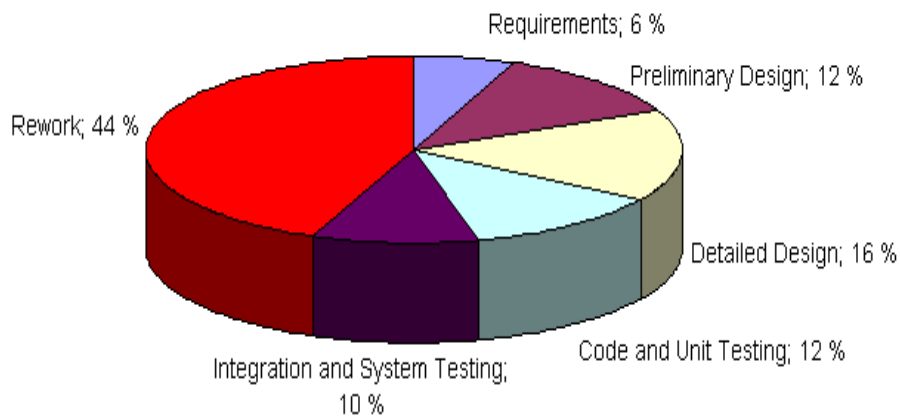


Figure 13 Cost of rework in software development

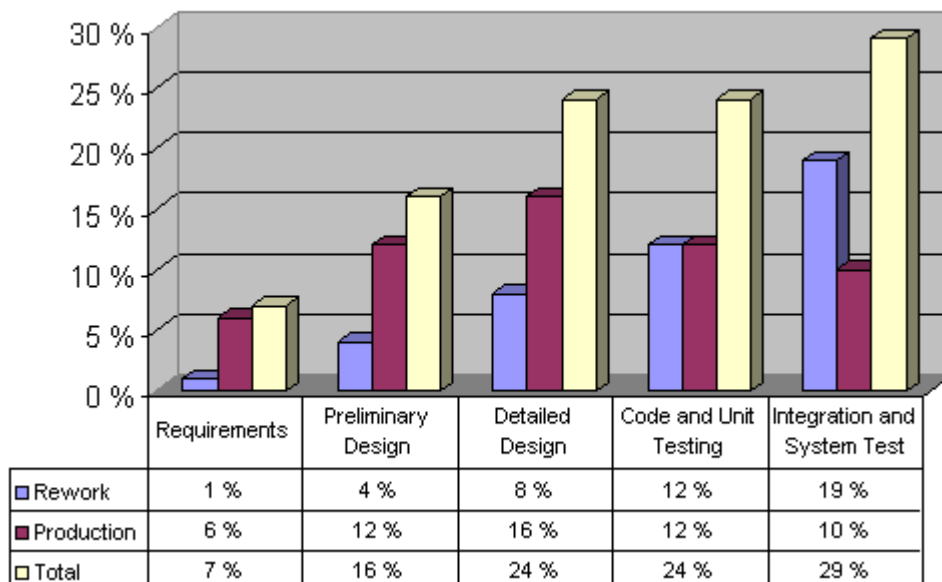


Figure 14 Percentage of rework in software development

Several studies have shown that the costs for defect detection and defect correction during inspections are much lower than other techniques, such as testing. According to Laitenberger et al, since inspection is a human-based activity, its costs are determined by human effort involved in inspection process [Laitenberger02]. The table gives a summary of results with respect to effort:

Project	Inspection's Result
IBM Santa Teresa Lab [Remus84]	Ratio of the cost of fixing defects during inspection to fixing them during formal testing is 1:20
Large real time software project [Collofello89]	7.5 hours per defect for design inspection, 6.3 hours per defect for code inspection, and 11.6 hours per defect for testing
Jet Propulsion Laboratory [Kelly92]	1.75 hours per defect of design inspection, 1.46 hours per defect for code inspection, 17 hours per defect for testing
Bull HN Information Systems [Weller93]	1.43 hours per defect in inspection and 6 hours in testing
Applicon [Gilb93]	0.9 hours to find and fix a major defect
IBM Rochester Lab [Kan95]	Ratio of the cost of fixing defects during inspection to fixing them during formal testing is 1:13
Jet Propulsion Laboratory [Kaner98]	Ratio of the cost of fixing defects during inspection to fixing them during formal testing range from 1:10 to 1:34

Table 3 Cost of inspections with respect to human effort

Project	Inspection's Result
Aetna Life Casualty [Fagan76]	82% of all detected defects
IBM Respond [Fagan86]	93% of all detected defects
Standard Bank of South Africa [Fagan86]	Over 50% of all detected defects
Large real time software project [Collofello89]	Effectiveness for design inspections: 54% Effectiveness for code inspections: 64% Effectiveness for testing: 38%
Bull HN Information Systems [Weller92]	70% of all detected defects
Hewlett-Packard [Grady94]	Over 60% of all detected defects
AT&T Bell Laboratories [Barnard94]	30%-75% of all detected defects
Cardiac Pacemakers Inc [McGibbon96]	70% to 90% of all detected defects
Erickson [Conradi99]	Average number of defects detected: 3.41

Table 4 Results of inspections with respect to quality

2.4. Summary

This chapter has provided an overview of the state-of-the-art of software inspections. The table given below summarizes the inspection methods discussed in this chapter.

Inspection Method	Team Size	Multiple Sessions	Meeting	Detection Method
Fagan	Large	No	Yes	Ad-hoc
FTARM	Large	No	Optional	Ad-hoc / Checklist
Gilb	Large	No	Yes	Checklist
Two-person	Small	No	Yes	Ad-hoc
Phased	Small	Yes	Yes	Checklist
N-fold	Small	Yes	Yes	Ad-hoc
ADR	Small	Yes	Yes	Ad-hoc
Meeting-less	Large	No	No	Any
Code reading (CR)	Small	No	Optional	Ad-hoc
CR by stepwise abstraction	Small	No	Optional	Ad-hoc

Table 5 Comparison of inspection techniques

This chapter has also presented the main points of the ‘Orthogonal Defect Classification Scheme’. We have also discussed that inspections can have a significant cost saving affect because defects are found early in the life cycle and less effort will be spent on debugging. It is a good way to enforce standards and provide necessary information for defect prevention activities such as root cause analysis.

We must avoid the common pit-falls that can undermine the efficiency of inspections. Participants should be adequately prepared for the group meeting. To ensure this, sufficient time should be given for individual preparations. To gain maximum benefit from inspections, the comments and criticisms should be directed to the product itself, not the author. Inspections should focus on defect detection, rather than defect correction. Selecting a good review rate is a vital factor that can significantly affect the efficiency of inspections.

3. Related Work

Not everything that counts can be counted and not everything that can be counted counts. – Albert Einstein

This chapter has the following sub-sections:

- **Section 3.1 ‘Related experiments’:** This section introduces the most relevant experiments that have been conducted in the software inspection field. The discussion focuses on the points that are relevant to our study.
- **Section 3.2 ‘Summary’:** This section gives a summary of the experiments discussed in the chapter.

3.1. Related Experiments

The section presents the experiments that are related to our experiment. It discusses 11 experiments that focus on aspects of the reviewing process. We found a large number of such experiments but we selected only the ones that are somehow related to the scope of our experiment i.e. reviewer's experience and defect classification schemes. There are experiments that do not directly discuss these issues but have been included in the section to show the variety of sources of variations that can affect our results.

The focus of this chapter is not to provide a summary of the experiments but to discuss the aspects that must be considered in our study. This includes discussion on the objects, the subjects and the experiment design of the study. The two most important variables discussed are the defect classification schemes and the subjects' ranking depending on their experience. The discussion reveals that most of the experiments either did not consider the affects of these variables or considered too many variables at the same time, which made it difficult to isolate and quantify the effects of these two variables. We believe that this information will guide us to plan a more effective experiment design.

3.1.1. Myers 78

The study was conducted at IBM Systems Research Institute. It compared three testing techniques; computer-based testing with access to program's specification, computer-based testing with program's specification and source code available, and walkthrough/inspection method. The main independent variable was the factors that influence tester's defect detection effectiveness and efficiency.

The subjects were 59 professionals from the computing industry with an average working experience of 11 years. They tested a text formatting PL/I program with 15 original and seeded defects. Each subject used only a single testing technique on the same program. The subjects were pre-tested and ranked according to their previous testing experience, proficiency in PL/I programming, and experience with walkthroughs/inspections. Based on the ranking, the subjects were divided into three main categories according to the testing technique used. The purpose of the ranking was to minimize the bias by the above variables and keep the average testing experience in each category at the same level.

The results showed no significant difference between the three testing techniques in detecting defects. Walkthrough/inspection method was found to be more expensive in terms of labour cost than other techniques. The variability in the types of detected defects was higher for computer-based testing than for walkthrough/inspection method, which makes walkthrough/inspection a more predictable testing method. All the subjects were highly experienced but there was still a great variation in the individual results, both in terms of the number and types of defects found. They could not correlate their results with the subject's experience due to great variability in individual performance. For example, in one case a subject who used to work as a technical writer found considerably more defects than a subject employed as a test specialist.

Another pattern in the results showed that walkthrough/inspection method focused heavily on the defect classes related to the program's logic. It was less effective in detecting types of defects that focus on the data handles by the program e.g. input and output data.

They also ran experiment sessions using different combinations of the above-mentioned three techniques. Two independent subjects who conducted testing and combined their results later were found to be more effective and efficient than any single technique. This was due to great variation in the types of detected defects between the subjects. See [Myer78] for further details.

3.1.2. Basili 87

This study compared three testing strategies; code reading by stepwise abstraction, functional testing, and structural testing. The main goal of the comparison was to investigate how the effectiveness of defect detection is affected by the tester's experience, defect classes, testing techniques and the interaction between these three factors. The experiment was conducted in three phases at the University of Maryland and NASA Goddard Space Flight Centre. The experiment involved 74 subjects where 32 of them were professional programmers and 42 were computer science students from different levels. The main research goal was divided into the following questions:

- Fault detection effectiveness:
 - Which testing technique detects the most faults in the code?
 - Are the numbers of detected faults dependent on the application domain?
 - Is the number of detected faults affected by the tester's experience?
- Fault detection cost:
 - Which testing technique has the highest defect detection rate with regard to fault-detection-effort?
 - Is this rate dependent on the application domain?
 - Is this rate dependent on the tester's experience?
- Classes of detected defects:
 - Do these techniques detect different classes of defects?
 - Are there any defect classes that can not be detected by these techniques?

The experiment used four programs from different application domains. There was a text processor, a mathematical plotting routine, a numeric abstract data type, and a database maintainer. Three of the programs were written in FORTRAN and one was written in Simple-T-Structured programming language. To create the worst-case scenario for the reviewers, the source code was totally uncommented.

The programs had in total 34 faults, including both original and seeded ones. The faults were classified into two main classes, omissions and commissions. The experiment used fractional factorial design with 222 testing sessions. Testing technique, application domain, and tester's experience were used as the independent variables. Each subject used all the testing techniques, one by one, but each on a different program.

In case of the professional programmers, code reading detected more defects and had better detection rate than functional testing and structural testing. In case of the students, the results varied from group to group. In one of the groups, code reading and functional testing detected more defects than structural testing. The fault detection rate was same for all the techniques. Code reading was more effective in detecting interface faults than other techniques, while functional testing detected more control faults than the other methods. Fault detection rate, fault-detection-effort, and the number of detected faults were dependent on the application domain of the program. See [Basili87] for further details.

3.1.3. Porter and Votta 94

The experiment compared Scenario-based reading (SBR) with Checklist-based reading (CBR) and adhoc reading. They hypothesized that systematic techniques such as SBR achieve higher defect detection rates and greater cost benefits than nonsystematic methods. Their hypothesis was based on the assumption that nonsystematic methods where each reviewer has general reviewing responsibilities will lower the overall effectiveness of defect detection due to the lack of coordination. Systematic approaches like SBR, where each reviewer executes a single scenario to detect particular classes of defects, will increase the coordination and overall effectiveness of the defect detection procedure.

The experiment used random, partial factorial design. It had four independent variables; the detection method, the team composition, the specification to be inspected, and the order in which the specifications were inspected. The only dependent variable was the team defect detection rate.

The experiment was multi-trial, with 24 graduate students used as subjects. The subjects were categorized into three categories depending on their background knowledge and experience, as low, medium, or high. Then they formed three person teams by randomly selecting individuals from each category. During the group meetings, each team member was assigned to act as the moderator, the recorder, or the reader. The study was conducted in two phases. The first phase was a training phase where the subjects were taught reviewing and experimental procedures. During the second phase, the subjects conducted two monitored inspections. The objects were three software requirements specifications: a water level monitoring system with 42 original defects, and an automobile cruise control system with 26 original defects, and an elevator control system with an unknown number of defects. No defects were seeded in any of the documents.

The defects were classified into two main types: omissions and commissions. The omission defects were subdivided into four categories: missing functionality, missing performance, missing environment, and missing interface. Commission defects were divided into ambiguous information, inconsistent information, incorrect or extra information and wrong section. The subjects were provided with a copy of the defect classification scheme. Both the individual defect detection and the collection meeting were limited to two hours each.

The results showed that the SBR was the most effective in defect detection and the checklist method was the least affective. The SBR was also found to increase the defect detection rate. For most of the defect classes, the reviewers using the SBR were more effective at finding the defects the scenario was designed to uncover. In case of the defects that were not addressed by the scenarios used in the SBR, all three techniques were equally affective at detecting these defects. See [Porter94] for the details.

3.1.4. Basili 96

The purpose of the study was to compare PBR technique's effectiveness in uncovering defects with the approach people were already using for reading and reviewing requirements specifications. The method assumed some experience in reading requirements documents on the part of the subjects. The subjects were classified as; designers, testers, and users, according to their experiences. The study had the following primary and secondary research questions:

- If groups of individuals (such as during an inspection meeting) were given unique PBR roles, would a larger collection of defects be detected than if each read the document in a similar way?
- If individuals read a document using PBR, would a different number of defects be found than if they read the document using their usual technique?
- Does a reviewer's experience in the role (designer, tester, and user) influence performance when using PBR?

Two classes of documents were used as the objects: a domain-specific set of documents for NASA, and a generic set that is more representative of other domains. For the first one, two small specifications (with 15 defects each) derived from an existing set of requirements documentation were used. The generic set of documents included an automated parking control system (27 seeded defects), and an automated bank teller machine (29 seeded defects).

They blocked the design on the reading technique, the subject's perspective, the requirements documents and the reading sequence. The subjects were 26 professional software developers from the NASA, Goddard Space Flight Center, and from the Software Engineering Laboratory. The experiment was conducted in two runs, the first one with 12 subjects and the second one with 14 subjects.

A series of partial factorial experiments was designed, in which subjects were given one document and told to discover defects using their current method. Then they were trained in PBR and given another document in order to see if their performance improved. Each subject had time to read and review no more than four documents: two from the generic domain, and two from the NASA domain.

The results indicated that the teams applying PBR achieved significantly better coverage of the documents than teams that did not apply PBR. The defect detection rates of teams applying PBR were also higher compared than for the teams using the usual technique. The results for the individual effectiveness showed that the defect detection rate for PBR reviewers was slightly higher than for reviewers using their usual technique. This represented a 21% improvement over the usual detection rate, but the difference was not statistically significant. See [Basili96] for the details.

3.1.5. Porter and Votta 97

The experiment was conducted on a real-life software project at AT&T Bell Labs. The main purpose was to compare the costs and benefits of making a variety of structural changes to the software inspection process. The study investigated the following questions:

- Are the effects of process structure obscured by other sources of variation? To address this question, the study attempted to separate the effects of some external sources of variation from the effects due to changes in the process structure.
- Are the effects of other factors more influential than the effects of process structure? The study compared the variation due to process structure with that due to other sources.

Based on these questions, the study hypothesized that:

- Inspections with large teams have longer intervals, but find no more defects than smaller teams.
- Multiple-session inspections are more effective than single-session inspection but at the cost of a longer interval.

This was a large-scale experiment during which 88 code inspections were performed over a period of 18 months. The experiment's object was a compiler for telephone switching system that contained over 55KLOC of C++ code. The subjects were 11 professional developers, where six developers had participated in the development of the compiler. They all had a similar development background and at least five years of experience.

The independent variables included: the number of reviewers (1, 2, or 4); the number of sessions (1 or 2); conduction of the multiple-sessions (parallel or sequence). The inspection effectiveness and interval were used as the dependent variables. The defect classification scheme was not shown to the reviewers of the authors to avoid biasing them.

The results showed that the inspection interval and effectiveness of defect detection were not affected by team size (small or large). It was not affected by the number of sessions (single or multiple) either.

The effectiveness of defect detection was not improved by performing repairs between sessions of two-session inspection; however, the inspection interval was significantly increased.

They examined how different reviewers affect the number of defects detected during the individual preparation. The results indicated that different reviewers found different number and types of defects because they were looking for different kinds of issues. For example, the development teams members raised on average more total issues but most of the issues were software maintenance issues, because, as authors of the project, they had a higher concern for its maintainability than the rest of the reviewers. See [Votta97] for the details.

3.1.6. Kelly 97

The study was conducted at the Queen's University and the Kingston Royal Military College in Canada. They compared task-directed reading, ad-hoc reading, and white-box testing. The reviewer's defect detection effectiveness was not measured by the number of defects found by them. The study differentiated between easy to find defects and more complex defects that require deeper understanding of the program. The effectiveness was measured by counting the number of "subtle" defects found by each participant. The Orthogonal Defect Classification was used to categorize the found defects by a defect type and a defect qualifier.

Graduate students were used as subjects and their educational background and industrial experience varied significantly. They used a repeated measured design and partial factorial design, where each subject used all the three testing techniques, one by one, on different portions of the code.

The source code used was written in Visual Basic. The code was already in use in the industry and has been under continuous development and maintenance during the last five years.

Unlike most of the other experiments, no defects were seeded into the code. Each category in the Orthogonal Defect Classification was related with a level of code's understanding that an inspector must attain to identify a defect.

The results showed a great variation between the subjects' effectiveness, depending on their experience. The results showed that the experienced inspectors using the task-directed techniques identified a higher proportion of findings that required deeper understanding of the code. They did not provide any statistically significant result, but it seemed like the defect detection tasks must be appropriate to the inspector's background and experience and require a minimum level of experience to be effective. See [Kelly97] for further details.

3.1.7. Emam and Wieczorek 98

This study evaluated the repeatability of a defect classification scheme using real code inspection data. The evaluated defect classification scheme was an adaptation of the Orthogonal Defect Classification (ODC) scheme, which had also been incorporated in the SEI's Personal Software Process. It used two data sets totaling 605 inspection defects. The method that they followed could be applied to evaluate the repeatability of other defect classification schemes using data that is commonly collected during software inspections.

The data that was used in the study came from a development project conducted within a company in Germany. The system consisted of approximately 30 KSLOC which was developed by a team of five developers. The application was a data analysis program that implemented a proprietary data mining technique. The code inspection process was restricted to two persons. During the individual preparation, the inspectors detected and classified the defects using a checklist. Each inspector filled out one defect report form that contained a defect's location, description and classification. After preparation, the two inspectors perform an inspection meeting.

Some of the defect classes from the ODC scheme were not included for the project under study, for example, timing/serialization defect type was removed since it was not applicable to the type of application. A number of extra defect classes were also added.

The distribution of defects for the first data set indicated that most of the defects that were logged during the meeting - almost 50% - were 'Documentation' type defects. This reflected that in that environment, documentation standards were not enforced consistently. There was a substantial difference in the distribution of all defects that were logged against those that are found by both inspectors, which reflected the specialization of the inspectors.

The study included only defects that were found by both inspectors during preparation. They had two data sets. In total, these two data sets represented 605 defects found during inspections. In the first data set only 23% of the defects were found by both inspectors and 24% in the second data set. This confirmed the fact that in that environment the inspectors specialize in finding different types of defects.

The results indicated that the classification scheme was in general repeatable. They evaluated classes of defects to find out if confusion between some categories was more common.

They suggested a potential improvement to the scheme. They suggested that the 'Data defect class be either refined further to clarify its distinguishing features from the 'Assignment' class, or merged with the 'Assignment' class. They concluded that the extent of agreement between two inspectors was sufficiently high, making the defect classification scheme reliability at least good and therefore was usable in practice without any modification. See [Emam98] for the details.

3.1.8. Laitenberger 98

The main objective of the experiment was to investigate whether code reading outperforms structural testing or vice versa. They also checked the combination of these two techniques and classes of defects found by them. The first hypothesis investigated whether code reading was more effective or efficient in defect detection than structural testing.

Another hypothesis was related to a scenario where a combination of code reading and structural testing was applied to check the percentage of defects that remain undetected after applying both techniques. They also investigated whether each testing technique favour any particular classes of defects to see if they complement each other.

The subjects were twenty graduate students, where only a few had any experience in software inspections. The source code used was 262 (excluding comments) lines of C-code that calculated statistical measures such as mean, standard deviation etc. Each subject was responsible for testing and reviewing 200 lines of codes. Unlike previous studies, the subject did not use these techniques on separate programs but on the same source code.

Code reading was conducted before structural testing and had three main phases; defect detection, defect collection, and defect correction. Two subjects participated in each inspection meeting. In structural testing, coverage values were used as a test criterion. The objective was to reach maximum (100% if possible) code coverage. Code coverage criteria included branch coverage, loop coverage, and relational coverage.

They seeded 13 defects in the source code. Defect detection technique was the only independent variable involved in the experiment. Dependent variables included defect detection effectiveness and efficiency of individuals and teams. The experiment was conducted on five consecutive days that also included training in inspection and testing techniques.

The inspection teams detected 52% of the defects, and only 17% of the remaining 48% defects were detected by the teams using structural testing. This showed that code reading was more effective and efficient than structural testing in detecting defects. They concluded that code reading and structural testing do not complement each other well when applied in sequence, since no evidence was found that code reading or structural testing detect defects of any particular class that are missed by the other one.

On average 39% of defects remained undetected and to achieve better coverage, they suggested to use code reading with some other testing technique as boundary value analysis. See [Laitenberger98] for further details.

3.1.9. Laitenberger 99

This paper compared one scenario-based reading technique, namely perspective-based reading (PBR), for defect detection in object-oriented design documents using the notation of the Unified Modeling Language (UML) to the more traditional checklist-based reading (CBR) approach. The experiment had the following null hypotheses:

- An inspection team is as effective or more effective using CBR than it is using PBR.
- An inspection team using PBR finds defects at the same or higher cost per defect than a team using CBR for the defect detection phase of the inspection.
- An inspection team using PBR finds defects at the same or higher cost per defect as a team using CBR for the meeting phase of the inspection.
- An inspection team using PBR finds defects at the same or higher cost per defect than a team using CBR for all phases of the inspection.

The subjects were 18 professionals with various levels of experience in object-oriented programming and UML. The average experience was two years, the minimum was one year and the maximum was sixteen years. Individual variability was controlled by random assignment of subjects to teams. No time constraints were imposed on the subjects, i.e., the subjects had as much time available as needed to complete their assigned tasks.

Two different systems were used as objects. One of them was a web-based quiz system which included six collaboration diagrams and four design class diagrams. The second system was a sales system with six collaboration diagrams and three design class diagrams. They seeded twenty one defects in the quiz system and nineteen defects in the point of sales system. Most of the defects were related to correctness, consistency, and completeness of the design models.

The experiment had four dependent variables; team defect detection effectiveness and the cost per defect, with three different definitions. The independent variables were the reading technique (CBR versus PBR) and the order of reading (CBR-PBR versus PBR-CBR). The experiment was conducted in four sessions, where each session lasted 0.5 days.

The results showed that the that PBR teams discovered, on average, 58% of the defects and had an average cost per defect ratio of 56 minutes per defect. The effectiveness of teams using PBR was 41% greater than of those using CBR, where the sequence of the various reading techniques did not influence the results. The teams using the PBR technique had a lower cost per defect ratio than the teams using the CBR technique, i.e., PBR exhibited, on average, a 58% cost per defect improvement over CBR. They claimed that the application of PBR scenarios improved the subject's perceived understanding of the specified system. See [Laitenberger99] for further details.

3.1.10. Basili 2001

The study was conducted by the researchers at the University of Maryland and the Fraunhofer Center-Maryland. This was the same group of researchers who had created the PBR techniques. The study was done in order to understand if there was any difference between inspectors who are familiar with a technique (PBR) and those who are using the technique for the first time. They believed that a better understanding of this type of knowledge was useful

because the effects of experience with an inspection process will help determine whether a novice or an expert is more effective. The study was concerned with two measures, effectiveness and efficiency. Effectiveness was measured as the percentage of known defects found in a software artifact, while efficiency was measured as the effort required in discovering a defect.

The goals were divided into three main classes; technical experience, domain knowledge, and software development experience:

- **Technique Experience:** The first GQM goal was to analyze PBR for the purpose of characterizing and understanding the impact of process experience with respect to effectiveness and efficiency from the point of view of the researcher. It addressed the following question:
 - How does it affect the effectiveness and efficiency of a subject to observe the use of PBR by someone else prior to using PBR himself or herself compared with an inspector who does not observe the use of PBR first?
- **Domain Knowledge:** The second GQM goal was to analyze PBR for the purpose of characterizing the effect of domain knowledge with respect to effectiveness and efficiency from the point of view of the researcher. It addressed the following question:
 - How does it affect the effectiveness and efficiency of a subject to have experience in a domain affect the efficiency of an inspector during an inspection?
- **Software Development Experience:** The last GQM goal was to analyze PBR for the purpose of characterizing the effect of software development experience with respect to effectiveness from the point of view of the researcher. It addressed the following question:
 - How does software development experience affect the effectiveness and efficiency of an inspector during an inspection?

The subjects were 26 graduate students from the University of Maryland enrolled in a graduate level Software Engineering class. The subjects were paired up into 13 pairs, with one subject acting as the executor (responsible for applying the procedure) and the other as the observer (responsible for recording observations about the application). Each pair performed two inspections, switching roles in between. This setup allowed all 26 subjects to perform a requirements inspection.

The objects were requirements documents from two different systems: one for a Loan Arranger (LA) system with 18 seeded defects, and one for an automated parking garage control system (PGCS), with 32 seeded defects. The LA system was responsible for organizing the loans held by a financial institution and bundling them for resale to investors. The PGCS was responsible for keeping track of how many open spaces there were in a parking garage and for keeping track of sales of both reserved (monthly) tickets as well as non-reserved (daily) tickets.

Each team consisted of two subjects that were either both highly experienced in software development or both inexperienced in software development. It was assumed that the LA domain was unfamiliar to the subject population and the PGCS domain was much more

familiar. Therefore, at least one member in each pair was knowledgeable in the PGCS domain and at least one member was not knowledgeable in the LA domain, and each member of the team was assigned to review a document to satisfy that constraint. Subjects were categorized based on their development experience and domain knowledge. Random pairings of the subjects were made to satisfy the above constraints. Before the study, subjects received training in the PBR techniques to be applied and the observational methods. Training in PBR was accomplished in a 60- minute class lecture.

The experiment design had a factorial design with two treatments. In the first treatment, roughly half of the teams inspected the LA requirements and the other half the PGCS requirements. After this inspection was complete, the team members switched roles, i.e. the process observer in the first inspection became the process executor for the second inspection. The teams also switched requirements documents, from LA to PGCS or vice-versa. All subjects used the User perspective of PBR in both treatments.

The analysis of the technical experience did not show a statistically significant difference between the subjects in the first inspection and those in the second inspection. The only statistically significant improvement from inspection 1 to inspection 2 was for subjects with low requirements experience inspecting the LA. Based on the qualitative data, there was an indication that the subjects performing the second inspection felt more comfortable with the technique and thought that they better understood the assigned procedure. It was hypothesized that for the observation of an inspection of a requirements document to be helpful, that inspection must be performed on an artifact when the inspectors have high domain knowledge.

It appeared that in most cases, the subjects in the second inspection were less efficient than those in the first inspection. The only cases where the efficiency improved were for low experience subjects who were inspecting the LA requirements and for high experience subjects inspecting the PGCS requirements. Both of these cases of improved efficiency were cases where the second inspectors also found more defects than the first inspectors. They concluded that the technique experience did not show an effect on efficiency.

When checking the effects of the domain knowledge, the results showed that the subjects inspecting the PGCS requirements on average found 13.5% of the defects, while the subjects inspecting the LA requirements on average found 17.5% of the defects, which is counter to what might have been expected. The results were not statistically significant. The average effort per defect for the PGCS requirements, where the subjects had high domain knowledge was 32.8 minutes, while for the LA, where the subjects had low domain knowledge, it was 53.4 minutes. This statistically significant difference showed that while domain knowledge did not really improve the effectiveness of the inspection, it does appear to improve the efficiency.

Software development experience did not help the subjects' performance during the inspection. For the second inspection and overall (both inspections together), the highly experienced subjects did significantly worse than the low experienced subjects. One possible explanation for this result was that PBR neutralized the effect of software development experience, i.e., the new technique improved the performance of the inexperienced subjects while hurting the performance of the more experienced subjects. Software development experience appeared to make subjects less efficient during the inspection.

In inspection 1 alone, inspection 2 alone, and both inspections taken together, the low experience subjects needed less effort to find each defect. The number of defects found by the low experience subjects was higher than that of the high experience subjects, which meant that spending about the same effort, the low experience subjects found more defects. See [Basili01] for the details.

3.1.11. Awan 2003

This experiment was conducted by us at the Norwegian University of Science and Technology. It was a part of “Software Engineering – Depth Study” during the fall of 2003. The study tried to answer the following questions:

- Is checklist-based reading more effective than ad-hoc reading? Why?
- Should the smaller teams use ad- hoc reading or checklist-based reading?
- Should the same reading technique be used regardless of the team size?
- What other sources of variations can affect the effectiveness of software inspections and how?

In our experiment, we had two independent variables. The first one was the defect detection technique used for individuals and teams. Three of the six experiment groups used checklists during individual defect detection, while the other three groups performed this task in an ad-hoc manner. The second independent variable was the team size. We measured only the effectiveness and did not measure the efficiency values for each step of the defect detection process.

The subjects were 20 students from NTNU. We had students from third, fourth and fifth year of a five years’ computer engineering study. We conducted the experiment in a single session, which included forty-five minutes individual review and forty-five minutes group meeting. All the subjects received the same training prior to the experiment. Since we had students with different number of years of study, each subject was required to fill in a form regarding their experience in programming and knowledge of software inspections. The form included details like number of semesters studied, commercial experience in programming, experience in conducting code inspections etc. The groups were composed as two groups with two members each, two groups with three members each, and two groups with five members each.

The inspected source code was part of a Java client-server application. The server’s source code contained 91 lines, including comment and blank lines. The client’s source code contained 39 lines of code, including comments and blank lines. We seeded 13 defects in 130 lines of source code. Our questions-based checklist used in the inspection was two pages long, with 12 sections and 47 questions.

Ten students conducted the individual review with the checklist, while other ten did not use it. The meeting involved the subjects working in groups of two, three or five reviewers. Similar to the individual reviews, only half of the groups used checklists during the meeting, while the other three groups performed this task in an ad-hoc manner. Each group member acted as a reviewer. The group members selected one member to act as the recorder in addition to being a reviewer. The recorder’s job was to log the defects detected by the group. The focus of the meeting was on discussing the results from the individual reviews, removing false positives and finding previously undetected defects.

The analysis of the collected data from the individual review activity showed that the checklist-based reading was 20% more effective than the ad-hoc reading in detecting true defects. The results confirmed the superiority of the checklist-based reading in effectiveness of defect detection during the individual review activity, which strongly supported our first hypothesis, HA1 (significance level < 0.05) that the checklist-based reading detects more defects than the ad-hoc method, when used in the individual review.

During the data analysis of the individual reviews, we observed some interesting patterns concerning the sources of variations. It seemed that some reviewers tend to ignore the complex sections of the checklist. The data patterns revealed that the length of the checklist had a significant affect on the effectiveness of inspections. However, deciding the optimal size of the checklist is not a trivial task. If the length is short, e.g., one page, it cannot address all the important classes of defects and the questions contained in it would not be specific. This can decrease the effectiveness of less experienced reviewers. If the length is more than one page, as we had a two-paged checklist, some reviewers might not be able to check the whole list, due to lack of time, or may be just due to lack of motivation.

We found that in case of group size two, the checklist-based reading detected 69% of true defects, while the ad-hoc reading detected only 54%, which makes the checklist-based reading 22% more effective than the ad-hoc reading. For group size three, the checklist-based reading detected 77% of true defects and the ad-hoc reading detected only 62%, which means that the checklist-based reading was 20% more effective.

The results supported our hypothesis that the checklist-based reading detects more defects than the ad-hoc method, when used in the interacting group review by small teams i.e. group size two and three. The observed significance level (0,052) was close enough to 0.05 and the trends in the observed data revealed superiority of the checklist-based method over the ad-hoc method, which is why we decided to reject the null hypothesis. Since the number of groups was small, it was not possible to make statistically significant decisions about the results

Each defect was detected by at least one individual using ad-hoc method, while none of the individuals using the checklist-based method detected two of the seeded defects. The reason for this shortcoming was that our checklist did not address these classes of defects; otherwise, the first defect was quite easy to notice. The second undetected defect was difficult to be detected by static review, which was why only one individual using the ad-hoc technique was able to find it and it might have been due to his experience with this class of defects.

For the larger groups, the ad-hoc reading was 15.4% more effective than the checklist-based reading. We had only two large groups with five members each, which was not enough to conduct any statistical analysis, but this explorative analysis, has given us the idea that the checklist-based method might be more important for the smaller teams than the larger ones. See [Awan03] for further details.

3.2. Summary

The table given below shows the summary of the experiments discussed in the previous section.

Author(s)	Focus	Defect classification used?	Reviewer's experience considered?
Myers 78	Computer-based testing vs. inspections	Yes	Yes
Basili 87	Structural testing vs. vs. functional testing vs. code reading	Yes	Yes
Porter and Votta 94	CBR vs. SBR vs. adhoc reading	Yes	Yes
Laitenberger 96	PBR vs. other reviewing methods	No	Yes
Porter and Votta 97	Structural changes in inspection process	Yes	No
Kelly 97	Task-directed reading vs. ad-hoc reading vs. white-box testing	Yes	No
Emam and Wiczorek 98	Repeatability of Orthogonal Defect classification	Yes	No
Laitenberger 98	Code reading vs. structural testing	Yes	No
Laitenberger 99	Perspective-based reading vs. checklist-based reading	No	No
Basili 01	Experienced reviewers vs. non-experienced reviewers using perspective-based reading	No	Yes
Awan 03	Checklist-based reading vs. ad-hoc reading and group size	No	Yes

4. Research Method

If we search long enough and hard enough, we will find rational rules that show us the best ways to build the best software – Pfleeger SL

This chapter introduces several types of research methods based on the theory given in [Trochim99] [Robson93] [Popper60] [Link4] [Wohlin00]. We explain how this study uses a combination of these research methods. It has the following sub-sections:

- **Section 4.1 ‘Types of research method’:** This section introduces a wide range of research methods and classifies these research methods according to their purpose, process and outcome.
- **Section 4.2 ‘Experiment process’:** This section describes the main phases of the experiment process followed in this study. The required inputs and generated outputs by each phase are also discussed in this section.

4.1. Types of Research Methods

In “The Nature of Engineering”, GFC Rogers explains the aims of technological research:

“The essence of technological investigation is that they are directed towards serving the process of designing and manufacturing or constructing particular things whose purpose has been clearly defined. We may wish to design a bridge that uses less material, build a dam that is safer, improve the efficiency of a power station, travel faster on the railways, and so on. A technological investigation is, in this sense, more prescribed than a scientific investigation. It is also more limited, in that it may end when it has led to an adequate solution of technical problem.”

The standards set by researchers for the verification of knowledge are higher in science than other fields. Research in all scientific fields invariably begins with a question in search of an answer [Popper60]. Research question can arise from a theory that yields a prediction, or from previous research findings that raise a new question or sometimes just from curiosity.

Evaluation is different from classic scientific research, as shown by the following table adapted from Isaac and Michael 1981/3:

	Research	Evaluation
Purpose	New knowledge	Product delivery
Outcome	Generalizable conclusions	Specific decisions
Value	Explanatory, predictive	Determining worth
Impetus	Curiosity	Needs and goals
Key event	Hypothesis testing	Assessing attainment
Classic methods	Experiment Correlation	Systems approach, Objectives approach
Discipline	Control, Manipulation	Program planning and evaluation
Criteria	Validity	Credibility, Fit between observed and expected
Types	Pure and applied	Formative and summative

Table 6 Comparison of research and evaluation

A research in software engineering can rely on a wide range of methods. Most of these research methods have as their goal to answer empirical questions through controlled experiment, but different questions call for different research methods, because the nature of a research question often constraints the methods that can be used to answer it. Research methods can be classified by its purpose, process and outcome [Popper60] [Link4].

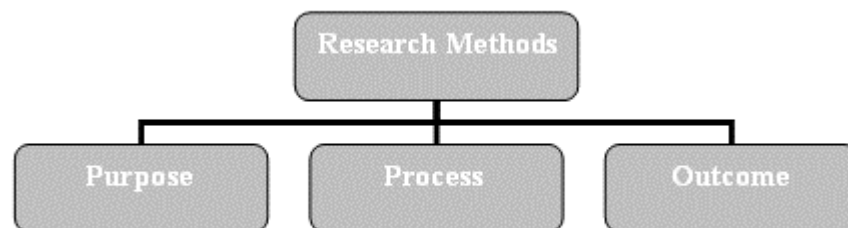


Figure 15 Three dimensions of the research methods

The purpose of a research can be exploratory, descriptive, analytical or predictive, as shown in Figure 16.

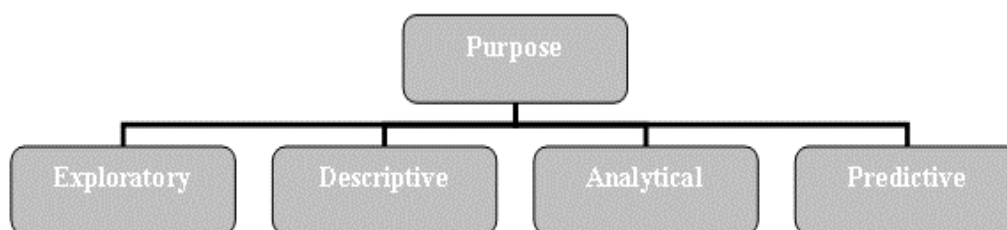


Figure 16 Types of research purposes

Exploratory studies lack representativeness and do not provide answers to research questions. It is used when there are no previous studies and the aim is to look for new ideas rather than testing old ones. It uses a qualitative approach to come up with new insights, which could lead to new research. The focus of this type of research is to gain insight into and familiarity with the subject area under investigation so that a more rigorous investigation can be conducted later.

Most of the time, the data from the descriptive research is used as an initial part of the research to describe the characteristics of the given population. The focus is on identifying and obtaining information on the characteristics of the situation under investigation.

Analytical research goes one step further than descriptive research and does not just describe the characteristics of a situation but also tries to analyze and explain why or how something is happening. The focus is on understanding phenomena by discovering and measuring causal relations among them.

Predictive research goes even further than analytical research and tries to forecast the likelihood of a similar situation occurring elsewhere. The focus is on generalising from the analysis by predicting certain phenomena on the basis of hypothesised, general relationships. The process of research can be either qualitative or quantitative, as shown in Figure 17 [Trochim99] [Link4]. The approach that should be taken is dependant on the purpose of the research. The main difference between these two approaches is related to the collected data during the research. Most of the research draws on both qualitative and quantitative methods since they are not always mutually exclusive.

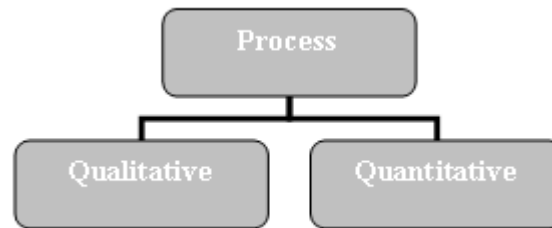


Figure 17 Types of research processes

Quantitative methods are concerned with quantifying a relationship or comparing two or more groups [Wohlin00]. The data is often analyzed with statistical methods where hypotheses are defined beforehand. The hypotheses propose a general relationship between variables. The three key methods that use quantitative approach are the following:

- **Survey:** This method involves asking representative samples of people standardized questions. The most common tools used for surveys include interviews and questionnaires. They are generally used to predict a future behavior, identify reactions to an event or examine relationship between constructs. Three main types of surveys include descriptive, explanatory, and explorative surveys. Questionnaires and interviews are generally used in the data collection phase of surveys.
- **Experiment:** This method measures the effects the manipulation of one variable has on another variable. It is the only method that can demonstrate causal relationship between variables. The subjects involved in the experiment are randomly assigned to the treatments. An experiment is called a quasi-experiment if the subjects are not randomly assigned to different treatments. See the next section for more details about the phases involved in experimentation.
- **Case study:** Case studies are most suitable for industrial software projects where the researcher will collect data from a single project within a given time period. The researcher can either compare the results with a standard baseline of the organization or select a sister project as a baseline. The researcher has no control over the project execution and it is thus hard to predict the nature of future phases of the project under investigation. The research is mostly of exploratory nature. Case studies are easy to conduct but the results are difficult to generalize.

The following table from [Wohlin00] compares qualitative and quantitative empirical strategies:

Investigation Method	Qualitative / Quantitative
Surveys	Both
Case Studies	Both
Experiments	Quantitative

Table 7 Qualitative vs. quantitative

The following table [Wohlin00] compares the research strategy factors of surveys, case studies and experiments:

Factor	Survey	Case Study	Experiment
Execution control	No	No	Yes
Measurement control	No	Yes	Yes
Investigation cost	Low	Medium	High
Ease of replication	High	Low	High

Table 8 Factors concerning research strategies

The outcome of the research can be classified as applied, basic or action [Trochim99] [Link4], as shown in Figure 18.

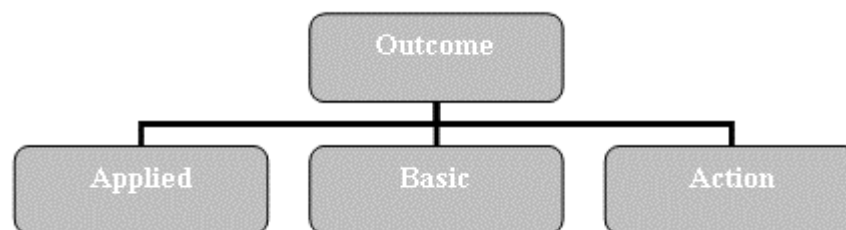


Figure 18 Types of research outcomes

Applied research is a problem-oriented approach where the research is conducted to solve a specific problem that generally requires a decision.

Basic research, also known as fundamental research, is carried out to improve our understanding of general issues. This kind of research does not emphasize immediate application. It can be further classified as discovery, invention or reflection. In the discovery form a new idea emerges from an empirical research that can have a significant affect on the current thinking on that particular topic. In the invention form, a new technique is created, while in the reflection form an existing theory is re-examined in a different context.

In action research, an action is both an outcome and part of the research. It changes what is being researched during the process of research. It can be either classical action research or new paradigm research. The classical action research is based on the idea that if a researcher wants to understand something, he should try changing it. The new paradigm research is based on a new framework for research in which the decisions are not made only by the researcher. The researcher acts as a member of a team. In this method the research is conducted as a mutual activity where those being investigated can also affect the decisions concerning how the research will be conducted.

Though our experiment has some elements of exploratory and descriptive research, it mainly falls into the analytical area. Unlike descriptive research, we will not only describe the characteristics of the situation under investigation but also analyze and explain why or how these things are happening.

As mentioned earlier, the main purpose will be to understand the phenomena by discovering and measuring causal relations among them. This will give us the opportunity to demonstrate the skills of analysis and evaluation. The process of our research is quantitative and will be conducted as a controlled experiment. The outcome of this research is characterised as 'basic research in the reflection mode'. See the next section for details about our experimentation process and its goals.

4.2. Experiment Process

Experimentation is used as the main research method for this study. Several authors have described the phases and activities involved in the experiment process [Wohlin00] [Juristo01]. This section gives a brief overview of the phases and activities that we plan to execute during this study. Figure 19 shows the main phases of the experiment process followed in this study and their required inputs and generated outputs.

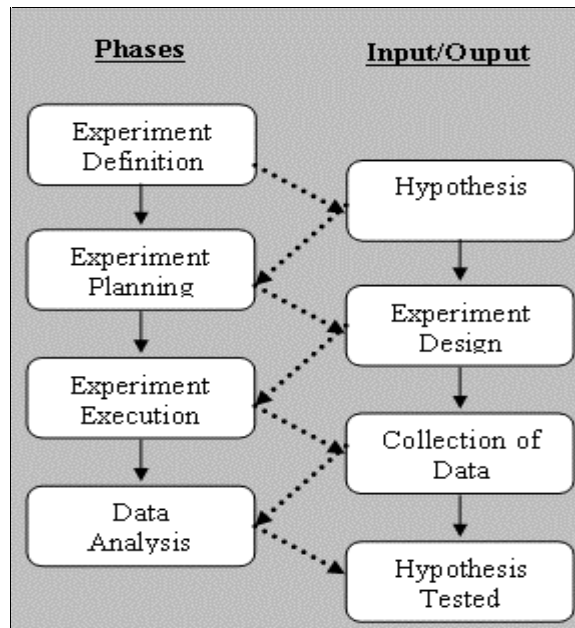


Figure 19 Phases of experimentation and their input/output

4.2.1. Experiment definition

This phase is entered once we have an idea for an experiment. The output of this phase is an informal hypothesis defined in terms of the variables of the phenomenon that will be observed. During the experiment definition, it is important to ensure that the defined hypothesis can be quantitatively evaluated so that the conclusion can be drawn from the results. The metrics for the quantitative evaluation are defined during the experiment design phase.

The experimenter selects the object of study, context, experiment purpose, focus and perspective of the experiment. The experiment context is defined in terms of the number of subject and object involved. The experiment is called a quasi-experiment if it lacks randomization of either subjects or objects.

We have used the Goal-Question-Metrics template for these definitions. Check chapter 4, “Experiment definition and planning” for more details.

4.2.2. Experiment planning

The input for this phase is the experiment definition and the output is the experiment design. It involves the following activities, as shown in Figure 20:

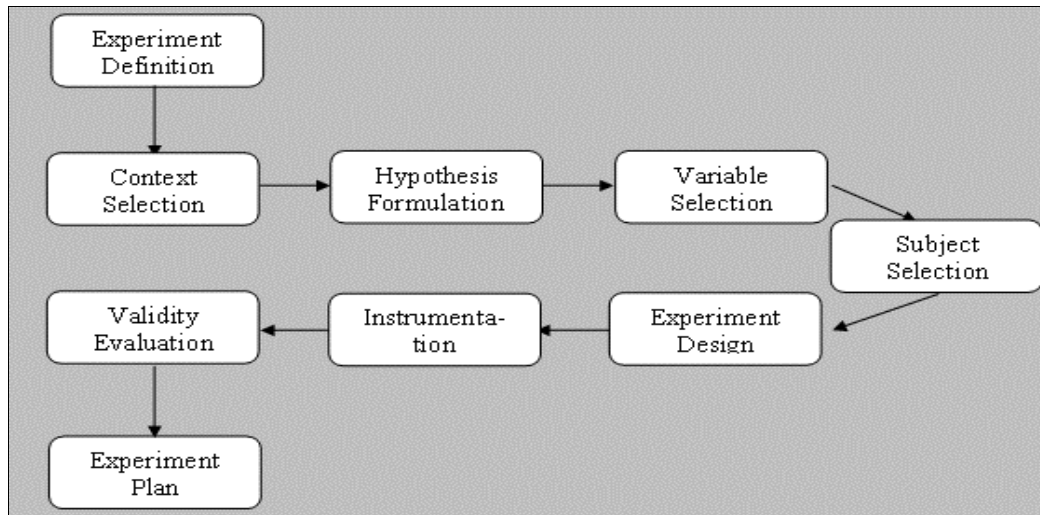


Figure 20 Experiment planning phase

The context of the experiment can be a real software project (on-line) or an alternative project (off-line) that is run parallel with a real project. It can either use students or professional developers as subjects to address a real or toy problem. In hypothesis formulation, a null hypothesis and an alternative hypothesis are specified. Risks that are involved in hypothesis testing are also considered e.g. Type-I-error and Type-II-error. Independent and dependent variables are selected. Selection of subjects can either use probabilistic or non-probabilistic sampling methods or we can use a combination. Choice of the experiment design depends on the number of factors and treatments used in the experiment. A good design should use randomization, blocking and balancing concepts to average out undesired effect of controlled or uncontrolled variables. In the instrumentation phase, objects, guidelines and measurement instruments are prepared e.g. source code, manual forms etc.

In the validity evaluation phase, the researcher considers the threats to conclusion, internal, construct and external validity. Low statistical power, fishing and the error rate, and random heterogeneity of subjects are examples of conclusion validity threats. Construct validity considers threats like mono-operation bias, mono-methods bias and hypothesis guess. History, maturation and statistical regression are examples of internal validity threats. External validity threats include interaction of selection-treatment, interaction of context-treatment, and interaction of history treatment.

More details about the experiment planning are given in chapter 4, “Experiment definition and planning”.

4.2.3. Experiment execution

The input for this phase is the experiment design and the output is the collected data from the experiment. It involves the following activities:

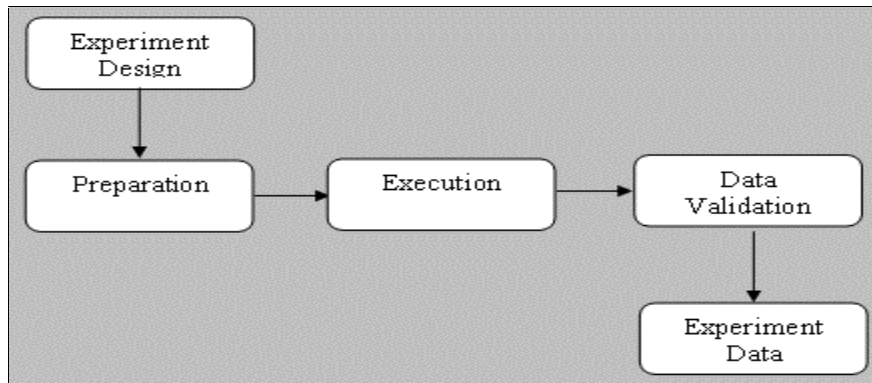


Figure 21 Experiment execution phase

The first activity is to prepare the necessary material for executing the experiment. It should be ensured that the subjects are committed to the whole experiment. Aspects such as obtaining the subject's consent, offering inducement for their participation, and confidentiality of their performance should also be considered. The research objectives should be introduced to the subjects to ensure that they agree to it, otherwise they might not participate in the optimal manner. All the instruments for execution e.g. manual forms, guidelines etc should be double-checked.

In the next step, the experiment is executed; the subjects perform their tasks as indicated by the selected experiment design and the experimenter collects the data. The data collection can either be done manually or automatically, depending on the experimental environment. If done manually, the experimenter should try to have a quick go through of all the forms and check that there are no inconsistencies. The subjects should not write their names on the forms, unless they agree to reveal their performance data to the others.

The last step is to validate the collected data to ensure that it is reasonable and that the subjects have participated seriously in the experiment. The experimenter should check that the subjects have conducted all the activities in the correct manner, as specified by the experiment design. If the subjects do not have a common understanding about the terminology used in the forms and other documents, the data might be invalid. If the forms can be backtracked to the subjects, the experimenter can discuss these problems and details about their data with the subject.

The experiment execution is the only phase where the subjects are actively involved and interact with the researcher. Most of the experiment guidelines and principles from the experimental field of psychology are also valid here and can be used effectively. The experiment can be easily replicated, if these guidelines are properly used. See chapter 5, "Experiment operation" for more details about how we executed the experiment.

4.2.4. Data Analysis and Interpretation

The input for this phase is the data collected in the previous phase and the output is the results of the hypothesis testing. The main activities involved in this phase are the following:

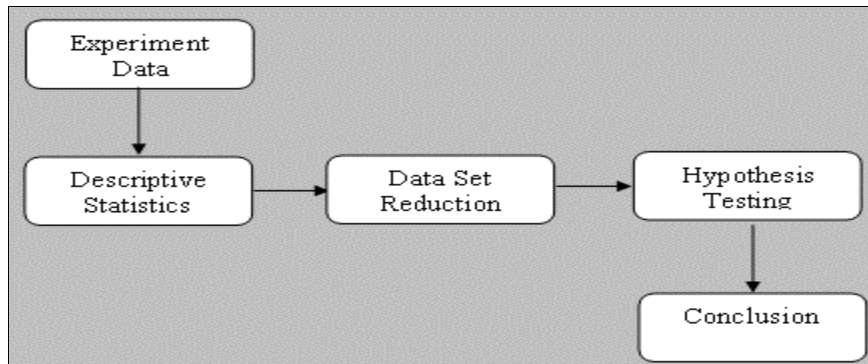


Figure 22 Analysis of the collected data

Descriptive statistics are used to characterize the collected data. It summarizes the collected data in a clear and understandable way. There are two main approaches, numerical and graphical. The numerical approach includes computing statistics such as mean, standard deviation, and variance. The graphical approach includes visualization tools such as histograms, box plots, and scatter plots. Numerical approaches are more useful in getting a precise answer, while graphical approaches are most helpful in identifying patterns in the data. The approaches complement each other and the researcher should use both of them.

The purpose of data set reduction is to reduce the data set by excluding the false data points. Graphical tools such as scatter plots and box plots can be used to detect the outliers. Whether the outliers should be excluded or not depends on the experiment design and other variables. Sometimes the outliers can be a source for generating ideas for the future research work and the experimenter should not ignore them, especially if the same kinds of outliers are detected when the experiment is replicated.

The next step is to analyze the reduced data by hypothesis testing. If the analysis reveals that the independent variable seems to have an effect, the researcher should be able to state with confidence that the effect was not just due to chance. Both the null hypothesis and the alternative hypothesis are tested to check if the null hypothesis can be rejected. There are two types of tests, parametric and non-parametric tests. Examples of parametric tests include t-test, F-test, and ANOVA. Chi-2, binomial test and Wilcoxon test are examples on non-parametric tests.

In the last step, the experimenter make conclusion based on the results of the hypothesis testing. Chapter 6 provides a detailed description of these activities together with the analysis and results of our experiment.

5. Experiment Definition and Planning

Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning. -Rich Cook

This chapter provides a formal description of the experiment. It includes the following sub-sections:

- **Section 5.1 ‘Experiment Definition’:** This section provides a formal definition of the experiment. The definition gives an overview of objectives, purpose and context of this study by using a GQM template.
- **Section 5.2 ‘Subjects’:** This section presents the background information of the subjects involved in the experiment.
- **Section 5.3 ‘Objects’:** This section gives a full description of the source code, defect classification system and the checklist used in the experiment.
- **Section 5.4 ‘Variables’:** This section defines the dependent and independent variables used in the experiment. It describes how these variables will be manipulated to generate the data that will help us to check our hypothesis.
- **Section 5.5 ‘Hypotheses’:** This section defines the hypotheses and the strategies that will be used to test them and a gives full description of the experiment design.
- **Section 5.6 ‘Experiment Design’:** This section gives full description of the experiment design.
- **Section 5.7 ‘Threats to Validity’:** The chapter ends with a discussion about the major validity threats faced by this study and how these threats are handled in the experiment.

5.1. Experiment Definition

We want to compare reviewers' effectiveness in defect detection, in a laboratory environment by selecting variables and controlling extraneous factors. Section 1.2 has given an overview of the reasons for conducting the experiment. We found that the effects of the reviewers' background on the efficacy of the individual inspection are still controversial. There is no definite answer to the relationship between the reviewers' experience and classes of defects detected by them. In spite of the extensive research done in software inspections' field, the following questions are still to be answered:

How important is the reviewers' experience in determining their effectiveness in defect detection during the individual review?

What is the relationship between the reviewers' experience and classes of defects detected by them?

Is there any uncontrolled variable that affects these two factors or interaction between them?

To define the goal definition of the experiment, we have used the Goal Question Metric (GQM). GQM is a systematic method for software measurement introduced by the University of Maryland [Basili90]. GQM focuses on the central role of a goal and reasons for measuring rather than a fixed set of metrics. It has three main stages. First, it sets goals in terms of purpose, perspective and context. Second, it refines the goals into questions that are quantifiable and traceable. Third, it answers the question from the second step by deducing the metrics and the collected data. These stages are shown in the figure given below [Link5]:

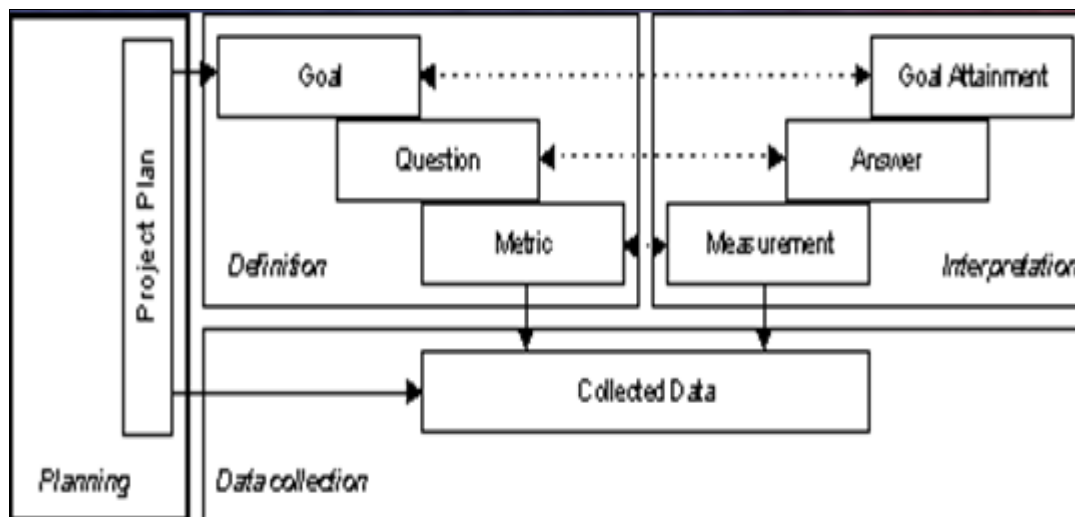


Figure 23 GQM Planning Phase

For the purpose of this experiment, we used the goal template structure defined by Basili and Rombach [Basili90]. It divides a goal into sub-components. The goal template is:

Analyze <Object(s) of study>
For the purpose of <Purpose>
With respect to their <Quality focus>
From the point of view of the <Perspective>
In the context of <Environment>

Objects of study: It identifies the entities studied in the experiment [Wohlin00]. In our case, the objects of the study are the reviewers' experience, classes of defects they detected and other sources of variations that could have direct or indirect affect on these two factors.

Purpose: This defines the main intention of the experiment [Wohlin00]. In our case, the main intention was to evaluate and characterize how the reviewers' experience can affect their effectiveness of code reviewing. In addition, we wanted to explore the inspection process to investigate if these two factors are affected by other sources of variations.

Quality focus: This defines the primary effect under study in the experiment [Wohlin00]. Our quality focus is defect detection effectiveness.

Perspective: This is the viewpoint from which the experiment results are interpreted [Wohlin00]. The perspective for our experiment is from a researcher viewpoint.

Context: This defines the environment in which the experiment is conducted [Wohlin00]. It also identifies the subjects and objects involved in the experiment. Our experiment involved students as the subjects, reading a small piece of Java code.

The experiment has the following goal definition:

- **Analyze**
 - the reviewers' experience and
 - classes of defects detected by the reviewers
- **For the purpose of**
 - evaluation
 - exploration
 - characterization
- **With respect to**
 - effectiveness
- **From the point of view of**
 - the researcher
- **In the context of**
 - undergraduate, graduate and PhD students reading 124 lines of Java code

	One object	> 1 object
One subject	Single object study	Multi-object variation study
> 1 subjects	Multi-test within object study	Blocked subject-object study

Table 9 Types of experiment contexts

[Wohlin00] has characterized the experiment context in terms of the number of objects and subjects participating as shown in Table 9. We have as single object study, when there is a single object and a single subject involved in the experiment. Multi-object variation study involves multiple objects and a single subject. Multi-test within object study, when there are multiple subjects and a single object involved in the experiment. We have as blocked subject-object study, when there are multiple objects and multiple subjects involved in the experiment. Our experiment is a multi-test within object study since we have several subjects working on one object (Java code).

5.2. Subjects

Subjects are the persons who will apply the methods to the experimental units. In our case, students are used as subjects. Unlike other fields like physics or chemistry, the results of software engineering empirical studies are dependent on the subjects involved in the experiment. Their selection must be done in a systematic manner to ensure that they represent the population of interest.

Sampling refers to how we select in sample from the target population. There are two main types of sampling methods: probability sampling and non- probability sampling.

In the probability sampling method, each subject in the population has a known non-zero probability of being included in the sample. The most important types of probability sampling methods are the following:

- **Simple sampling:** Selects n subjects randomly from a population of N . The method is useful for small populations because of its simplicity. It is not suitable for large populations because listing all the items in a large population will be a time-consuming task.
- **Systematic sampling:** Select every n th subject from a population of N . It is also known as interval sampling because there is an interval between each selection. The advantage is that it is easier to select every n th item from a population than selecting as many random items as the required sample size.
- **Cluster sampling:** Used to break up large groups into smaller clusters. First it randomly selects a number of clusters and then all the items from those clusters are selected to represent the population. It does not include any item from the non-selected clusters. For a same sample size, cluster sampling provides less accurate results than simple sampling.
- **Stratified sampling:** Divides the population into strata, based on some substantive definition. First we divide the population into groups and then we select samples from each group. Unlike cluster sampling, some items are selected from each group which ensures that we do not accidentally miss out a particular group from the population.
- **Multi-stage sampling:** This method combines other sampling methods which makes it more complex. The method involves two main stages. First, like cluster sampling, it divides the population into smaller clusters and then unlike cluster sampling it does not include all the items from a cluster but only selects a sample from them. It is more economic and efficient. However, like cluster sampling, this method suffers from lower accuracy due to higher sampling error.

The non-probability sampling methods are less structured and subject selection is left to the discretion of the researcher. It includes the following methods:

- **Quota sampling:** Subjects are selected on the basis of some identified characteristics. It can be either proportional or non-proportional. In proportional quota sampling, a proportional amount of each quota is sampled to represent the major characteristics of the population. In non-proportional sampling, the minimum number of the items from each category is specified. It is a non-probability alternative to the stratified sampling, since it ensures that the sample adequately represents the smaller groups.

- **Snowball sampling:** First find initial subjects who meet the criteria for inclusion in the study and then ask them for references to other potential subjects. This method does not guarantee a representative sample. It might be useful in situations where the researcher has limited access to the population.
- **Convenience sampling:** Uses the most readily available subjects. It must be applied with a great caution to ensure a representative sample.
- **Purposive sampling:** Researcher handpicks subjects to participate in the study based on identified variables. The researcher must take care in not overweighting readily accessible subgroups.
- **Modal instance sampling:** It uses the concept of “mode or typical case”, that is the most frequent value in the population’s distribution. Selection of the attributes that characterizes the “typical” case can be a challenging task, which makes this method useful only for informal sampling situations.

In the experiment, we used a two-stage process which was a combination of quota and convenience non-probability sampling methods. In the first stage, we divided the population into the following quotas, according to their educational background:

- Students from the second year
- Students from the third year
- Students from the fourth year
- Students from the fifth year
- PhD students

In the second stage, we used convenience sampling to select the subjects that are representative of their sub-group. In this way, we selected students with various educational backgrounds and experiences, both with and without any industrial experiment. We divided the students into two groups; highly-experienced students and less-experienced students. The students were placed into these groups based on their educational background, software development experience and formal software inspection experience.

The subjects were 42 students from the Norwegian University of Science and Technology (NTNU). All the subjects had taken one or more programming and software engineering courses. As a part of their degree, they had obtained practical training in software development where Java was the major language used. Most of the senior students had participated in a number of industrial software development projects that involved 5-6 developers working in a team.

To some extent, the senior students are comparable to fresh graduates working as professional software developers in industry. The junior students had less (or no) experience with industrial software projects and most of them had never participated in a formal software inspection. It was necessary to involve such subjects, since the difference of experience among the subjects was the main ingredient of the experiment

5.3. Objects

5.3.1. Java Source Code

The source code to-be-inspected and its documentation was a part of a Java client-server application, taken from [Link6]. It has three classes; *EncryptionClient*, *EncryptionServer*, and *EncryptionServerThread*. The multi-threaded server encrypts the text given by the client and returns it to the client. The server's source code contained 83 lines, including comment and blank lines. The client's source code contained 51 lines of code, including comments and blank lines. We seeded 12 defects in 124 lines of source code.

The *EncryptionClient* class implements the client program that speaks to the *EncryptionServer*. When you start the client program, the server should already be running and listening to the port, waiting for a client to request a connection. The first thing the client program does is to open a socket that is connected to the server running on the hostname and port specified:

```
try {
    // open up the connection on the given port
    socket = new Socket("encryptionHost", 4444);
    // prepare input/output streams to the server
    out = new PrintWriter(socket.getOutputStream(), true);
    in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
} catch (UnknownHostException e) {
    System.err.println("Don't know about host: encryptionHost");
    System.exit(1);
}
```

The *EncryptionClient* program also specifies the port number 4444 when creating its socket. This is a remote port number, the number of a port on the server machine, and is the port to which *EncryptionServer* is listening.

Next comes the while loop that implements the communication between the client and the server. The server speaks first, so the client must listen first. The client does this by reading from the input stream attached to the socket. If the server does speak, it says "Bye." and the client exits the loop. Otherwise, the client displays the text to the standard output and then reads the response from the user, who types into the standard input. After the user types a carriage return, the client sends the text to the server through the output stream attached to the socket.

```
// get input from the user via command line, until the server sends "Bye"
BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
while ((fromServer = in.readLine()) != null) {
    System.out.println("Server: " + fromServer);
    if (fromServer.equals("Bye.")) break;
    fromUser = stdIn.readLine();
    if (fromUser != null) {
        System.out.println("Client: " + fromUser);
        out.println(fromUser);
    }
}
```

The communication ends when the server says "Bye.", which means the client cannot use this encryption service anymore. In the interest of good housekeeping, the client closes its input and output streams and the socket:

```
// cleanup
out.close();
in.close();
stdin.close();
socket.close();
```

The server program begins by creating a new `ServerSocket` object to listen on a port 4444. `ServerSocket` is a `java.net` class that provides a system-independent implementation of the server side of a client/server socket connection. The constructor for `ServerSocket` throws an exception if it can't listen on the specified port (for example, the port is already being used). In this case, the *EncryptionServer* has no choice but to exit.

```
try {
    // open the socket
    serverSocket = new ServerSocket(4444);
} catch (IOException e) {
    System.err.println("Could not listen on port: 4444.");
    System.exit(-1);
}
```

If the server successfully connects to its port, then the `ServerSocket` object is successfully created and the server continues to the next step, accepting a connection from a client. *EncryptionServer* loops forever, listening for client connection requests on a `ServerSocket`. When a request comes in, *EncryptionServer* accepts the connection, creates a new *EncryptionServer* Thread object to process it, hands it the socket returned from `accept`, and starts the thread. Then the server goes back to listening for connection requests. The *EncryptionServerThread* object communicates with the client by reading from and writing to the socket.

```
try {
    // open the socket
    serverSocket = new ServerSocket(4444);
} catch (IOException e) {
    System.err.println("Could not listen on port: 4444.");
    System.exit(-1);
}
// keep running the server
while (listening) new EncryptionServerThread(serverSocket.accept()).start();
serverSocket.close();
```

After the server successfully establishes a connection with a client, it communicates with the client, first initiating the conversation. The `readLine` method waits until the client responds by writing something to its output stream (the server's input stream). When the client responds, the server passes the client's response to the *EncryptionServerThread* object and asks the *EncryptionServerThread* object for the encrypted data. The server immediately sends the reply to the client via the output stream connected to the socket, using a call to `println`. If the server's response generated from the *SecretEncryptor* object is "Bye.", the loop quits.

```
PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
BufferedReader in = new BufferedReader(
    new InputStreamReader(socket.getInputStream()));
String inputLine, outputLine;

out.println("Welcome to the encryption server");
while ((inputLine = in.readLine()) != null) {
    outputLine = "output>>" + SecretEncryptor.encrypt(inputLine);
    out.println(outputLine);
    if (outputLine.equals("Bye")) break;
}
out.close();
in.close();
socket.close();
```

The last several lines of this section of *EncryptionServerThread* clean up by closing all of the input and output streams, the client socket, and the server socket.

5.3.2. Defect Classes

Defect classification categorizes a defect in terms of its seriousness and its impact on software quality. A variety of defect classification schemes have been suggested by researchers. A defect can be characterized in terms of a commission or omission of statements, phase of software cycle where the defect was injected or discovered etc.

We decided to classify defects in the following three main classes as given in chapter 2:

- **Wrong statements:** When a code statement is not implemented in the correct manner. It can include logical errors, incorrect functional description etc.
- **Extra statements:** Software would function properly even if these statements are removed the source code e.g. unnecessary comments, declaring a variable that is never used etc.
- **Missing statements:** Examples of these defects include a code defect where error-checking is omitted, forgetting to close an open socket connection etc.

We conducted a single experiment session with 124 lines of codes and 12 seeded defects. This classification scheme enabled us to have a high level distinction between defect classes and injection of 5-6 defects from each defect class. This would not have been feasible with 6-7 categories of defects because then we would have only 1-2 seeded defect per category. We would not have been able to draw any sensible and statistically significant conclusion based on such data.

It should be noted that the defect classification was not shown to the subjects. They did not classify the defects into classes; this was done by us during the data interpretation phase.

5.3.3. Checklist

Checklists are an important tool for the reviewers, especially less-experienced ones. It plays a vital role in both individual and group activities of the reviewing process. In the individual reviews, it draws reviewer's attention to specific parts of the software artifact that is defect prone and provides important clues. During the group activities, it identifies the focus areas for group discussions and helps in structuring the review meeting. Checklist-based reading is seen as the standard reading technique in many software organizations [Thelin03]. It has often been used as a baseline technique in many experiments comparing different reading techniques.

Checklist-based reading offers more support to the reviewers than ad-hoc reading, but has its shortcomings. The major disadvantage is that checklist composition is highly dependant on past defect information in an organization and if no such information is available, the checklist is composed of the questions and issues given in the literature [Chernak96]. In such situations, the reviewers do not pay attention to previously undetected defect types and might miss whole classes of defects [Laitenberger00]. If a checklist contains too many questions, the reviewers can be swamped with too much information and unnecessary details [Parnas87].

Many researchers have confirmed that the use of checklist-based method is superior to the ad-hoc method. It was also confirmed by the experiment we conducted during the previous semester.

The checklist can be written as statements or questions. Our questions-based checklist used in the inspection was two pages long. It had 12 sections and 47 questions. The checklist's composition is shown in Table 11. The full checklist is given in Appendix C.

Section name	# questions
Variable, Attribute, and Constant Declaration Defects	8
Method Definition Defects	5
Class Definition Defects	3
Data Reference Defects	2
Computation/Numeric Defects	3
Comparison/Relational Defects	3
Control Flow Defects	8
Input-Output Defects	4
Module Interface Defects	2
Comment Defects	5
Layout and Packaging Defects	2
Exceptions Handling	2

Table 10 # questions in each section of the checklist

5.4. Variables

We want to gain knowledge by discovering relationships between the variables observed in the experiment. Three levels of relationships among variables can be interesting for the experiment; descriptive, correlation and causal relationship.

In descriptive relationship, we can describe behavioral patterns among variables but can not predict the occurrence of the behavior since the relation is unknown. Correlation relationship can be described using a function but cause and effect can not be distinguished. Causal relationship is the strongest level where the relationship among variables is deterministic. Probabilistic causality is better suited for software engineering, which is not 100% deterministic but enables us to predict the likelihood of causal relationship.

We have defined two types of variables for the experiment, independent and dependent variables. The main idea is to control the independent variables and measure their effect on the dependent variables.

5.4.1. Independent Variables

The term independent variable is taken from mathematics. These variables are changeable and controllable in the study setting [Wohlin00]. They are manipulated to generate the data that will help us to check our hypothesis. They are also called predictor variables since they are used to predict the influence on the values of the dependent variables.

The reviewers' educational background and experience was the only independent variable in our experiment. We made a short questionnaire to find out about their background and used a grading scheme to make two different classes, depending on their experience. Some of the questions are given below.

- How many semesters have you studied?
- How many programming or software engineering subjects have you taken?
- In how many software development projects (with at least three members in the team) have you participated?
- How much experience do you have in programming (any language)?
- How much experience do you have in Java programming?
- How much industrial experience do you have?
- Have you participated in a code review before?

The students were divided into two experience profiles, highly-experienced and less-experienced, depending on their response to the given questionnaire.

5.4.2. Dependent Variables

Dependent variables are the variables whose values vary in a predictable manner when manipulating the independent variables [Perry00]. They are also called response variables since they represent the outcome of an experiment. Unlike the independent variables, the dependent variables are simply measured and not manipulated by the experimenter.

The selection of measurement scale and range of the variables is related to the dependent variables. Gathering the data for each dependent variable is called an observation, and these observations are analyzed in the interpretation phase to test the validity of the hypothesis. We selected the following dependent variables:

- Number of class 1 defects found by each reviewer
- Number of class 2 defects found by each reviewer
- Number of class 3 defects found by each reviewer
- Number of defects from each defect class found by the reviewers in the junior profile
- Number of defects from each defect class found by the reviewers in the experienced profile
- Total defects found by each reviewer

The measures for effectiveness are a variation of “Review yield” proposed by Humphrey [Humphrey95]. This measure refers to the percentage of defects in the source code at the time of the defect detection activity. Syntax errors reported by the individual reviewers were not considered as true defects because the code was already compilable without errors and any reports of syntax errors would be a non-true defect.

We did not measure the efficiency values for each step of the defect detection process. Since individuals and teams had a fixed time limit, the effort differences would have been marginal.

5.5. Hypotheses

The purpose of formalizing the experiment definition into a hypothesis is to state clearly what we intend to evaluate in the experiment. The dependent variables are analyzed to evaluate the quantifiable hypothesis in the experiment.

Hypothesis testing is a method of inferential statistics. The null hypothesis is normally the reverse of what the researcher believes and hopes to reject it. The symbol H_0 is used to indicate the null hypothesis, and H_A is used to indicate the alternative hypothesis. H_A is always the opposite of H_0 .

Given that the collected data provides statistically significant results, H_0 can be rejected in favor of the alternative hypothesis. It should be noted that the failure to reject the H_0 is not similar to accepting H_0 . There can be results where the H_0 can neither be rejected nor accepted.

The reasons for selecting the following hypothesis are discussed in section 1.2:

- H_{A1} : *The highly-experienced reviewers detect more defects than the less-experienced reviewers, when using the checklist-based reading in the individual review.*
- H_{01} : *The highly-experienced reviewers do not detect more defects than the less-experienced reviewers, when using the checklist-based reading in the individual review.*

- H_{A2} : *The highly-experienced reviewers detect more ‘missing code statement’¹ defects than the less-experienced reviewers, when using the checklist-based reading in the individual review.*
- H_{02} : *The highly-experienced reviewers do not detect more ‘missing code statement’ defects than the less-experienced reviewers, when using the checklist-based reading in the individual review.*

- H_{A3} : *The highly-experienced reviewers detect more ‘extra code statement’ defects than the less-experienced reviewers, when using the checklist-based reading in the individual review.*
- H_{03} : *The highly-experienced reviewers do not detect more ‘extra code statement’ defects than the less-experienced reviewers, when using the checklist-based reading in the individual review.*

- H_{A4} : *The highly-experienced reviewers detect more ‘wrong code statement’ defects than the less-experienced reviewers, when using the checklist-based reading in the individual review.*
- H_{04} : *The highly-experienced reviewers do not detect more ‘wrong code statement’ defects than the less-experienced reviewers, when using the checklist-based reading in the individual review.*

¹ See section 4.3.2, Defect classes

The significance level is the criterion used for rejecting H_0 and shows that how likely a result is due to chance. In this experiment we used the 0.05 level. Statistical significance has to do with mathematical probability and it does not imply whether a research results are useful or not. Table 11 shows two types of errors related to the significance testing.

	H_0 true	H_0 false
H_0 rejected	Type-I-error	Ok
H_0 not rejected	Ok	Type-II-error

Table 11 Statistical decision and type of errors

Type-I-error is the probability of rejecting H_0 , when H_0 is true and should not have been rejected. Type-II-error is the probability of not rejecting H_0 , when H_0 false and should have been rejected.

5.6. Experiment Design

The experimental design is the phase where the researcher arranges the real world (“pre-treatment of the real world) before observing it [Juristo01]. The selection of a suitable design for an experiment depends on many factors e.g. the objectives of the experiment, the number of treatments etc. The experiment design also determines the selection of statistical analyses to be used during the data interpretation phase. The experiment design should use design principles such as randomization, blocking and balancing to average out the effect of undesired factors. In [Wohlin00], experiments are classified into four categories:

- **One factor with two treatments:** These types of experiments compare two treatments against each other. Two examples of experiment designs are complete randomized design and randomized paired comparison design. The first one compares two treatment means, where each subject uses a single treatment on one object. In the second design, the subject uses both treatments on the same object. This method has a disadvantage related to the learning effect since the subject uses the same technique twice and becomes more acquainted with this technique after the first time.
- **One factor with more than two treatments:** This includes simple randomized design and randomized complete block design. The first design uses one object for all the treatments where the subject-treatment assignment is done in a random manner. The second design blocks the experiment on the subjects to minimize the effect of variability between them.
- **Two factors with two treatments:** This includes $2 * 2$ factorial designs and two-stage nested designs. Factorial designs examine the effects of various combinations of different treatment factors by randomly assigning subjects to each combination of the treatments. The two-stage nested design is a hierarchical design where one factor is nested under the second factor.
- **More than two factors each with two treatments:** It includes 2^k factorial design and 2^k fractional factorial design. These designs use all the combinations and alternatives of all the factors, which cover the effect of every factor as well as their interactions with each other. Since the number of possible combinations grows rapidly, the fractional factorial designs can be used to leave out some of the combinations.

Our experiment had one factor with two treatments, since we had two student profiles depending on their experience. All subjects were required to inspect the same client-server Java application. The subjects inspected the code individually; there was no group activity involved. The forms filled in during the individual review were collected at the end of individual activities. For each defect found, the subjects noted the line number, Java class name and a short description. First, we divided the individuals into different profiles according to their experience and number of semesters studied and used systematic assignment to allocate the individuals to one of the two profiles. The purpose was to ensure that the experience and knowledge inside each group was at the same level. In this manner, we avoided the impact of lurking variables and ensured that any differences in performance between individuals in the same profile were not due to pre-existing differences between them.

5.7. Threats to Validity

One of the fundamental questions to be asked about any experiment is the validity and reliability of the study and collected data. Is the study designed and conducted in a controlled manner? Can we generalize the results? Is there any relationship between the treatment and the results?

In [Cook79], the threats to validity have been divided into the following categories:

- Conclusion validity
- Internal validity
- Construct validity
- External validity

For further discussion about how these threats correlate to experiment design and principles, see [Wohlin00] and [Trochin99].

There is no known way of developing a perfect empirical study, at least not with the limited resources that we had during this study. Nevertheless, our goal was to remove most of these threats or at least be aware of the limitations caused by their presence, which in turn helped us to make informed decisions about the best way to reduce their effects.

5.7.1. Conclusion Validity

Cook and Campbell have defined the conclusion validity as the extent to which conclusions are statistically valid [Cook79]. We controlled the conclusion validity by using robust statistical techniques and by ensuring that measures and treatment implementation are reliable. The threats concerning the statistical techniques used in the experiment are considered being under control. Threats with respect to the subjects are limited since subjects in each profile are rather homogeneous and attended the same education programs.

5.7.2. Internal Validity

Internal validity describes the extent to which the experiment design shows a cause-effect relationship between the dependent and independent variables [Campbell63]. In [Link4] and [Link5], several sources of threats to internal validity are described. The most important sources are the following:

Selection: This threat is caused by variations in human performance, as each subject bring unique characteristics with her into the experiment, such as proficiency in a programming language, etc. In our experiment, this threat was controlled by capturing the experience of our subjects.

Maturation: This threat is due to the changing of the subjects' behavior in the course of the experiment. Examples of such changes include learning effects, motivation factor, fatigue effects etc. It might be that these factors are causing the observed effect rather than the treatment that is being studied. The overall maturation threat was made negligible by doing the following:

- Giving similar training to the subjects to ensure that they all started at the same level.
- Not showing the source code to the subjects before the inspection, to ensure that they do not discuss the code before the experiment.
- Collecting the individual forms immediately after the individual review to prevent any changes, once the time limit is over.

History: History can pose a threat to internal validity when the subjects experience an event which is unrelated to the treatment and which has an impact on their performance on the post-test. We did not face any such event in our experiment.

Instrumentation: This threat arises when an observed effect is caused by a difference in the way the pre-test and post-test are measured, rather than the impact of the treatment that was implemented. We avoided this threat by proper use of statistical techniques and fulfilling their necessary assumption.

Experimental Mortality: This threat means that the subjects drop out of the study between the pre-test and the post-test, and an observed effect may be caused by the fact that the make-up of the group is not the same at both stages of measurement. This threat was not relevant for our experiment because none of the subjects dropped out of the study.

Repeated Instrumentation: This threat rises when the prior measurement of the dependent variable may affect the results obtained from subsequent measurements. For example, a situation in which a group may perform better on a post-test, not because of the treatment that was implemented, but because the pre-test primed the group members to perform better. This threat was not relevant for our experiment because our experiment was conducted in a single session and no repeated measurements were involved.

Regression to the mean: This threat relates to the observation that the subjects with extreme scores on a first measure of the dependent variable tend to have scores closer to the mean on a second measure. This threat was not relevant for our single session experiment.

5.7.3. Construct Validity

Construct validity refers to the extent to which independent and dependent variables map to the hypotheses [Jude91]. Mono-operation bias and mono-method bias are examples of the threats. Mono-operation bias pertains to independent variables and treatments in the study e.g. a single program is used in a single session, without any replication. Mono-method bias pertains to measures or outcomes e.g. a single measure of key construct is used. Examples of social threats to construct validity include hypothesis guessing, researcher expectancies, and evaluation apprehension.

As a first step to assess the construct validity of our experiment, we rigorously analyzed whether our selected metrics really capture the attributes of interest and are correlated with our intended effect as assumed. The performance measures used by us e.g. rate of faults found, are considered reliable. We believe future replications of the experiment would further limit these threats.

5.7.4. External Validity

External validity concerns the ability to generalize the results to industrial practice [Campbell63] [Jude91]. According to [Wohlin00], there are three main risks:

- Subjects are not representatives of their field and population-of-interest
- Experimental environment is not realistic i.e. the documents used are not representative of real-world industrial practice
- The timing for conducting the experiment can affect the results

The largest threat to the external validity in our experiment is the use of students as subjects. Since most of the students in junior profile had none or little industrial experience, they may not be representative of professional software engineers. We have not regarded this threat as being critical, because the main intention of the experiment was to involve both junior and senior students. On the other hand, most of the senior students were currently in their last years of study and will start their professional life quite soon.

Many researches have investigated the issue of using students as subjects in empirical studies. We have seen arguments both for and against this practice. [WohlinRegnell00] has investigated whether there are differences of using students or professional software engineers in empirical studies and found no significant difference between their efficacies of finding defects.

Using students as subjects can also be useful when students are used to demonstrate the initial validity of a research idea to enable industry to have sufficient confidence in the research to justify their participation in follow-up research [Ritchey2001].

The code used in this experiment may not be representative of real-world code with regard to its size and complexity. The Java code that we used was only 130 lines, while most of the real-world applications are much larger. However, the time spent on the inspection of this code module was consistent with that suggested in [Fagan76]. According to Fagan, the total time of inspection should not exceed two hours and the inspection rate should not be more than 125 LOC per hour, otherwise their efficiency in detecting errors might start falling. So even software inspections in real-world large projects must necessarily be broken down into smaller tasks, and this experiment mirrors adequately such small subtasks.

6. Experiment Operation

*True research is like fumbling in the dark for the right switches.
Once you have turned the light on everyone can see... Unknown*

The chapter describes how the experiment was prepared. It includes descriptions of aspects that will ease replication of the experiment and gives an insight into how activities had been carried out. It presents the execution of the experiment and explains how the data was collected during the experiment. It also provides a case for the validity of the data. It presents the factors that could have affected the experiment operation. It has the following sections:

- **Section 6.1 ‘Conducted Activities’:** This section presents the main activities that we conducted during the experiment. It has the following subsections:
 - **Overview and training:** This subsection explains how we trained the subjects for the experiment. It explains how we presented the goals of individual defect detection to the subjects.
 - **Individual defect detection:** This subsection describes the execution of the individual review activity. It also explains how we collected the data during the review.
 - **Data validation:** This subsection stresses the validation procedures of the data collection during the experiment.

6.1. Conducted Activities

We conducted three main activities during the experiment:

- Overview and training
- Individual defect detection
- Data validation

Since we had students with different number of years of study, each subject was required to fill a form regarding their experience in programming and knowledge of software inspections. The form included details like number of semesters studied, commercial experience in programming, experience in conducting code inspections etc. During the whole experiment, we encouraged the subjects to feel free and raise any questions regarding any technical or non-technical details. The individual defect detection required the completion of an individual form. These forms were used to record the description and location of the defects found.

6.1.1. Overview and Training

We provided all the subjects with the same training prior to the experiment, in the form of a short presentation. The presentation introduced the activities involved in the inspection process during the experiment, the inspection materials and instructions on how to conduct the required activities. We explained the goals of individual defect detection and the specific procedures to be used in the experiment. We also presented the forms that were used during the individual and group defect detection. It included the experience form, individual defect detection form. We instructed the subjects to find as many find defects as possible, and to avoid doing any solution hunting for the detected defects.

6.1.2 Individual defect detection

This activity lasted for 45 minutes. We provided a package of the necessary material to each subject, which included the source code to-be-inspected, the inspection checklist and the individual forms for logging the defects found. For each defect found, the reviewers logged the line number in the source code that contained the defect, together with a short description of the defect. The subjects independently reviewed the source code, with the use of the checklist.

Figure 24 shows the input and output of the individual review activity. It also shows the factors that can influence the efficacy of this activity. For details about these factors and their classification, see [Armbrust02].

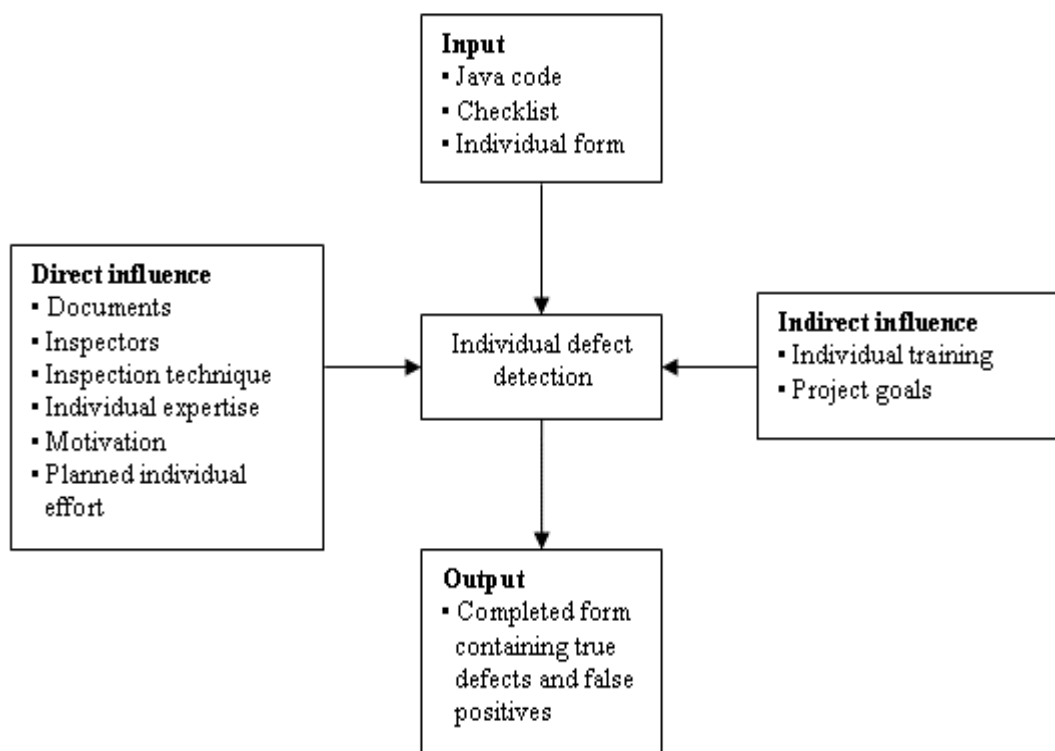


Figure 24 Individual review activity and the factors that affect its effectiveness

The direct factors can greatly affect the inspection process. The list given below describes the most important direct factors:

- **Documents:** It has been proven that the document under inspection can significantly influence the efficacy of the inspection process [Votta98]. The size of the document to-be-inspected is important. If it is a large document, then there is a risk that reviewers would skip through some parts of the document and concentrate on the selected parts, which they expect to have high density of defects. If the document to-be-inspected is small, they have time to read the entire document and then concentrate on their selected parts. The reviewers would not be able to conduct an effective defect detection, if the document to-be-inspected is complex or difficult to read and understand. In our case, it was a small document and the reviewers had sufficient time to conduct an effective individual defect detection activity.
- **Inspectors and their expertise:** The effectiveness and efficiency of the individual defect detection activity depends heavily on the reviewers' experience, technical expertise and domain knowledge [Votta98]. These key points determines how well the individual defect detection is conducted and how thoroughly the document will be inspected. There is no doubt that the expertise of the subjects involved in our experiment affected the results. We have tried to control this affect by documenting their experience.

- **Inspection technique:** Some inspection techniques are more effective in detecting defects of particular defect classes and directly influence the inspection process. In our experiment, all the subjects used checklists in individual defect detection activity.
- **Motivation:** Motivation to find maximum number of defects in the document under inspection differs from reviewer to reviewer and significantly affects their effectiveness in detecting defects. It is hard to quantify this variable and very little research has been done in this area.

6.1.3. Data Validation

The purpose of this activity was to ensure that the collected data is valid and the data collection activity was performed correctly. We checked that all the subjects had filled in the forms correctly and had the same understanding of the contents of these forms. We also checked that the subjects had applied the correct techniques and treatments in the intended order.

7. Data Analysis and Interpretation

It is not primarily the responsibility of a statistician to make decisions for other people — not in general, at any rate . . . It is for someone else to say what decisions should be made with [inferential] . . . information. In other words, ideally, it is the statistician's job to inform not to decide. Kerridge, D. F. Journal of the Royal Statistical Society.

This chapter provides a presentation of the data analysis. It describes the calculations together with the assumptions for using the selected analysis models and provides an interpretation of the data and explains the reasons for rejecting or accepting the hypothesis. It includes information about the sample sizes and significance levels of the results and discusses the most important patterns observed in the data. It includes the following sub-sections:

- **Section 7.1 ‘Data Analysis’:** This section presents the data analysis for our hypotheses concerning the effectiveness of individual review activity. First it present descriptive statistics to discover the differences between the two groups and then provides a more in-depth analysis with inferential statistics (t-test).
- **Section 7.2 ‘Data Interpretation’:** This section presents interpretation of the results found in the previous section and discusses the implications of these results.

7.1. Data Analysis

Personal expertise and inspection's experience is an important source of variation in individual defect detection activity. Regardless of the reading technique used for defect detection, the effectiveness of individual defect detection depends heavily on the experience of the reviewers. In the experiment we divided the subjects into two main groups, depending on their education and experience; highly-experienced subjects (HE) and less-experienced subjects (LE). For the details about the experiment design see chapter 4.

The following hypothesis considers the overall defect detection effectiveness of the reviewers:

H_{A1} : *The highly-experienced reviewers detect more defects than the less-experienced reviewers, when using the checklist-based reading in the individual review.*

H_{01} : *The highly-experienced reviewers do not detect more defects than the less-experienced reviewers, when using the checklist-based reading in the individual review.*

Figure 25 gives an overview of the detected defects and compares the effectiveness of the HE subjects with the LE subjects.

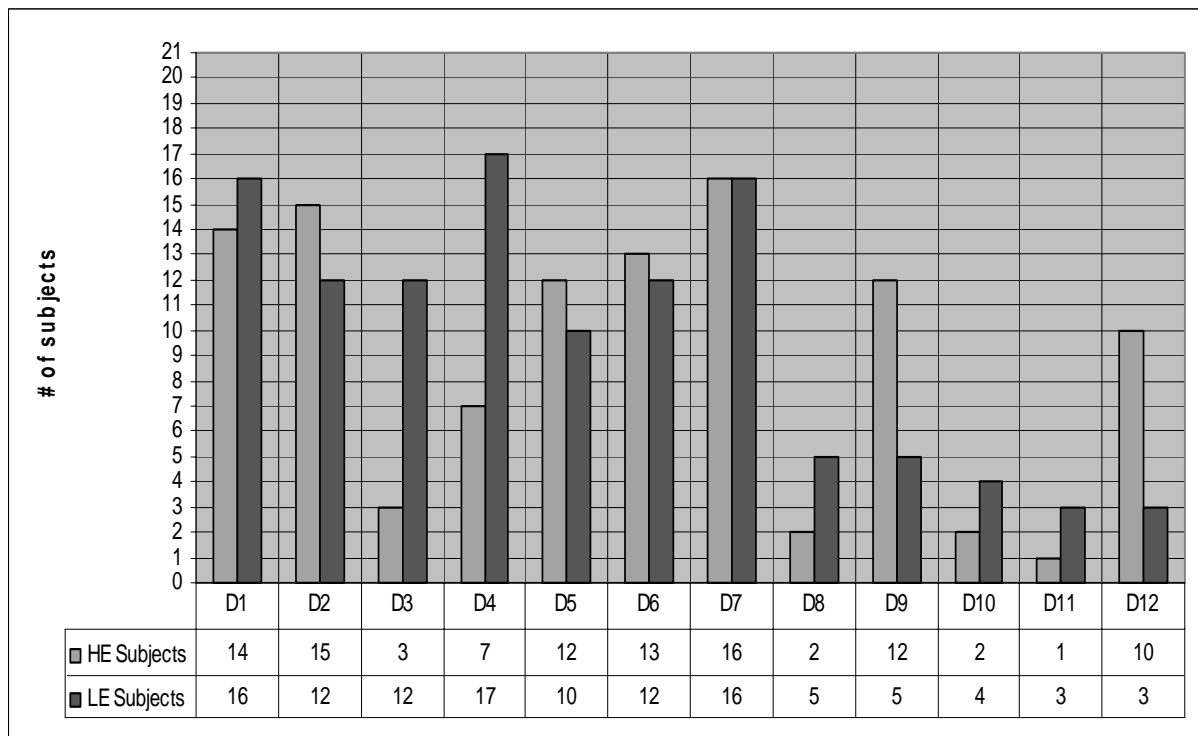


Figure 25 compares the number of subjects in each group which detected the given defects (D=Defect)

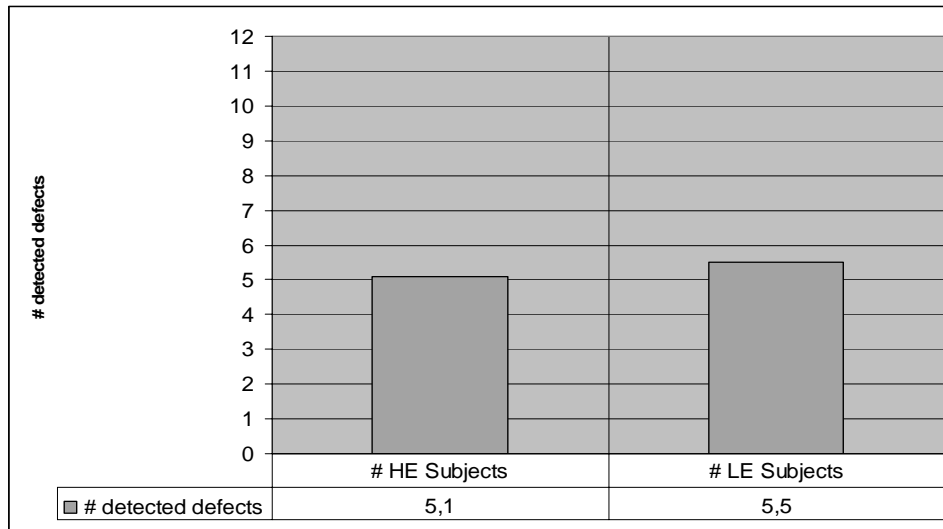


Figure 26 Comparison of effectiveness in defect detection (Mean values)

Figure 26 shows a comparison of the effectiveness of highly-experienced (HE) subjects with less-experienced (LE) subjects, when using the checklist-based reading during the individual defect detection. We had seeded 12 defects in the source code. Syntax errors reported by the individual reviewers were not considered as true defects because the code was already compilable without errors and any reports of syntax errors would obviously be a non-true defect.

The HE subjects detected 42.42% of true defects on average, while the LE subjects detected 45.58%. The LE subjects were slightly more effective than the HE subjects in detecting true defects. Although the results did not show the superiority of the HE subjects to the LE subjects, Figure 25 indicates that different subject groups found different number and types of defects because they might be looking for different kinds of issues, depending on their experience and educational background. We will discuss these issues later in this chapter.

We have used the t-test to see if it is possible to reject our null hypothesis, H_0 , based on the observations. The t-test is a parametric test, which is generally used to check the differences between the means of two groups. It calculates the difference between their means relative to the variability of their values.

There are two main types of the t-test, paired sample t-test and two-sample t-test. In two-sample t-test there is one test that assumes equal variances between samples and another that is used when sample variances are unequal. If individual data points in sample 1 are paired with individual data points in sample 2 then the paired sample t-test should be used. A two-sample t-test is used when individual data points are unpaired.

In our case, we used the two-sample t-test that does not assume equal variances between samples. In two-sample tests, the size of sample 1 need not equal the size of sample 2, but in our case, they were equal. The p-level reported with the t-test represents the probability of error associated with rejecting the hypothesis of no difference between the two categories of observations in the population when, in fact the hypothesis is true. In our case, p was set to 5%.

The degree of freedom must also be determined for the test. The result of the t-test is greater than zero if the first mean is larger than the second mean. If the result is a negative number then the first mean is smaller than the second one.

We applied the t-test to the data given in Table 12. HE stands for the highly-experienced subjects and LE stands for the less-experienced subjects.

Student ID	True defects	Student ID	True defects
HE-1	1	LE-1	4
HE-2	5	LE-2	6
HE-3	6	LE-3	5
HE-4	1	LE-4	3
HE-5	7	LE-5	4
HE-6	4	LE-6	5
HE-7	4	LE-7	4
HE-8	7	LE-8	7
HE-9	5	LE-9	6
HE-10	8	LE-10	6
HE-11	5	LE-11	5
HE-12	7	LE-12	6
HE-13	9	LE-13	4
HE-14	5	LE-14	3
HE-15	4	LE-15	3
HE-16	3	LE-16	5
HE-17	3	LE-17	6
HE-18	5	LE-18	7
HE-19	7	LE-19	11
HE-20	6	LE-20	9
HE-21	5	LE-21	6

Table 12 # of defects found by the HE (highly-experienced) and the LE (less-experienced) subjects

Table 13 shows the result of the t-test. The data shown in the table is for one-tail t-test, which does not support our hypothesis that the mean for sample one (HE subjects) is greater than the mean for sample two (LE subjects). We are specifying the direction of difference rather than that they are just different.

t-Test: Two-Sample Assuming Unequal Variances		
	Highly-experienced subjects (HE)	Less-experienced subjects (LE)
Mean	5,09	5,47
Variance	4,29	3,86
Observations	21	21
Hypothesized Mean Difference	0	
Degree of Freedom (df)	40	
t Stat	-0,61	
P(T<=t) one-tail	0,27	
t Critical one-tail	1,68	

Table 13 t-Test conducted on the defects data (12 defects)

If the first mean was larger than the second mean, t-Stat would have been greater than zero. But in this case the result is a negative number, which means that the first mean is smaller than the second one. ‘P (T<=t) one tail’ is the probability of getting a t-Stat that is greater than or equal to the one we got. According to the results, we can not reject our null hypothesis and accept, H_0 , that the experienced-reviewers do not detect more defects than the less-experienced reviewers, when using the checklist-based reading in the individual review.

Considering the trends that the data has shown, we turned our hypothesis the other way around and assumed that the less-experienced reviewers actually detect more defects than the highly- experienced reviewers, and conducted statistical analysis on the same data. All the data given in the table given above remained same except the value of t-Stat, which was a positive number (0.61) in this case. Although the results showed a minor superiority of the LE subjects in overall defect detection effectiveness, which is counter to what might have been expected, the results were not statistically significant.

Although we could not find any statistically significant difference in defect detection effectiveness between different groups of the subjects, there were some patterns in the data that revealed a relationship between the reviewers’ experience and the classes of defects detected by them. The second hypothesis given below, addresses one of these defect classes:

H_{A2} : *The highly-experienced reviewers detect more ‘missing code statement’ defects than the less-experienced reviewers, when using the checklist-based reading in the individual review.*

H_{02} : *The highly-experienced reviewers do not detect more ‘missing code statement’ defects than the less-experienced reviewers, when using the checklist-based reading in the individual review.*

We seeded twelve defects in the source code, where four of them were from ‘missing code statement’ defects class e.g. error-checking is missing, forgetting to close an open socket connection etc. In the source code, D3, D4, D8, and D10 belonged to this defect class. The figure given below compares the number of the subjects that found these defects.

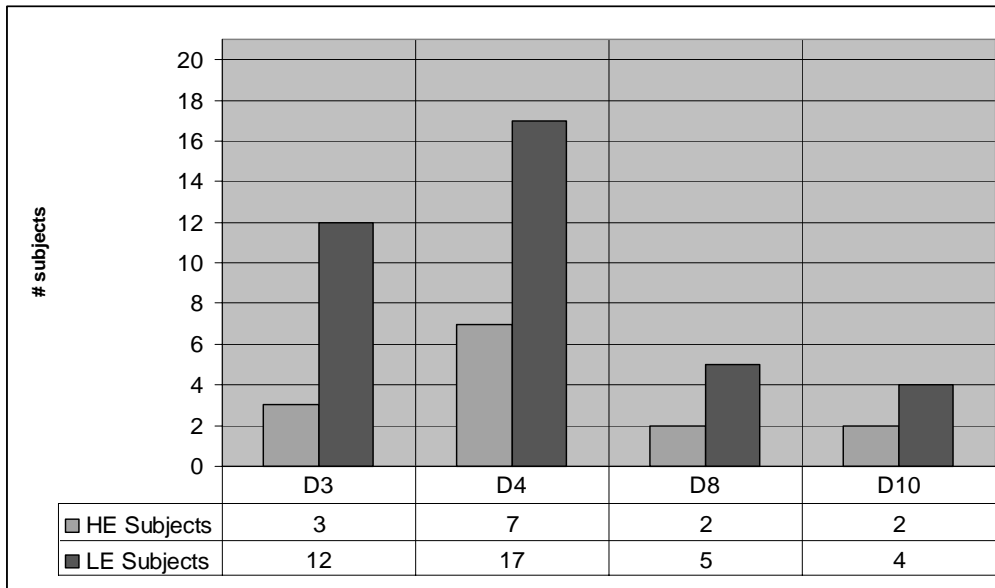


Figure 27 'Missing statement' defects detected by the subjects

The above figure shows that the less-experienced (LE) subjects did considerably better than the more-experienced (HE) subjects, in detecting the 'missing code statement' defects. On average, the HE subjects detected 0.67 and the LE subjects detected 1.81 of the four seeded 'missing code statement' defects.

We used the t-Test to test the second hypothesis and found the following results:

t-Test: Two-Sample Assuming Unequal Variances		
	Highly-experienced subjects (HE)	Less-experienced subjects (LE)
Mean	0,67	1,81
Variance	0,43	0,86
Observations	21,00	21,00
Hypothesized Mean Difference	0,00	
Degree of Freedom (df)	36,00	
t Stat	-4,60	
P(T<=t) one-tail	0,00	
t Critical one-tail	1,69	

Table 14 t-Test conducted on 4 missing code statement defects

As shown in the table, the first mean is smaller than the second one, and the resulting t-Stat is a negative number, -4.60. Thus, we can not reject the null hypothesis, H_0 , that the highly-experienced reviewers do not detect more 'missing code statement' defects than the less-experienced reviewers, when using the checklist-based reading in the individual review.

In the same manner as we did with our first hypothesis, we can turn the hypothesis the other way around and assume that the less-experienced subjects detect more 'missing code statement' defects than the highly-experienced subjects.

In this case, the value of the t-Stat is a positive number (4.60) and the value of the ‘P (T<=t) one-tail’ (0.00) is less than 0.05. The t-Stat value (4.60) is also greater than ‘t Critical one-tail’ (1.69), we can thus be 95% sure that the mean for sample 1 (LE) is greater than the mean for sample 2 (HE). We have statistically significant results to reject the null hypothesis and can thus conclude that the less-experienced subjects detect more ‘missing code statement’ defects than the highly-experienced subjects.

The next hypothesis addresses the ‘extra code statement’ defects.

H_{A3} : *The highly-experienced reviewers detect more ‘extra code statement’ defects than the less-experienced reviewers, when using the checklist-based reading in the individual review.*

H_{03} : *The highly-experienced reviewers do not detect more ‘extra code statement’ defects than the less-experienced reviewers, when using the checklist-based reading in the individual review.*

Four of the seeded defects in the source code belonged to the ‘extra code statement’ defects. It refers to the defects when the software would function properly even if these statements are removed the source code e.g. unnecessary comments, declaring a variable that is never used etc. In the source code, these defects are D2, D5, D9, and D12. The figure given below compares the number of the subjects that found these defects.

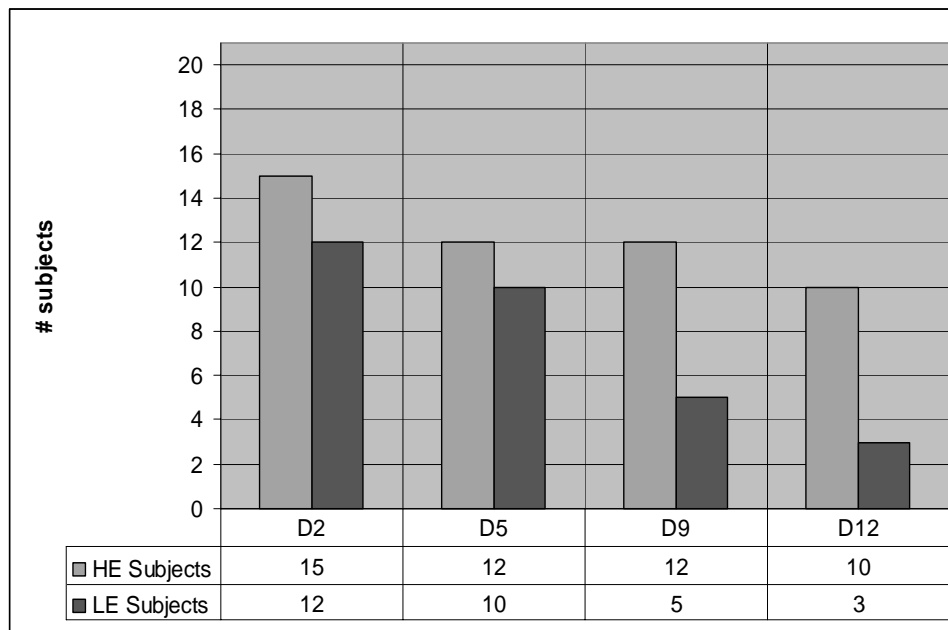


Figure 28 ‘Extra statement’ defects detected by the subjects

As shown in the figure, unlike other defect classes, the HE subjects were more effective than the LE subjects in case of ‘extra code statement’ defects. We conducted the t-Test for testing the hypothesis and found the following results:

t-Test: Two-Sample Assuming Unequal Variances		
	Highly-experienced Subjects (HE)	Less-experienced subjects(LE)
Mean	2,33	1,43
Variance	2,23	0,86
Observations	21,00	21,00
Hypothesized Mean Difference	0,00	
Degree of Freedom (df)	33,00	
t Stat	2,36	
P(T<=t) one-tail	0,01	
t Critical one-tail	1,69	

Table 15 t-Test conducted on 4 extra code statement defects

The value of the t-Stat is a positive number (2.36) and the value of the ‘P (T<=t) one-tail’ (0.01) is less than 0.05. The t-Stat value (2.36) is also greater than t-Critical one-tail’ (1.69), we can thus be 95% sure that the mean for sample 1 (HE) is greater than the mean for sample 2 (LE). We can thus reject the null hypothesis, H_03 , that the HE subjects do not detect more ‘extra code statements’ than the LE subjects. The next hypothesis addresses the ‘wrong code statement’ defects.

H_{A4} : *The experienced-reviewers detect more ‘wrong code statement’ defects than the less-experienced reviewers, when using the checklist-based reading in the individual review.*

H_{04} : *The experienced- reviewers do not detect more ‘wrong code statement’ defects than the less-experienced reviewers, when using the checklist-based reading in the individual review.*

The figure given below compares the effectiveness of the HE and the LE subjects in detecting these defects. The ‘wrong code statement’ defects are generated when a code statement is not implemented in the correct manner e.g. logical errors, incorrect functional description etc. D1, D6, D7, and D11 belonged to this defect class.

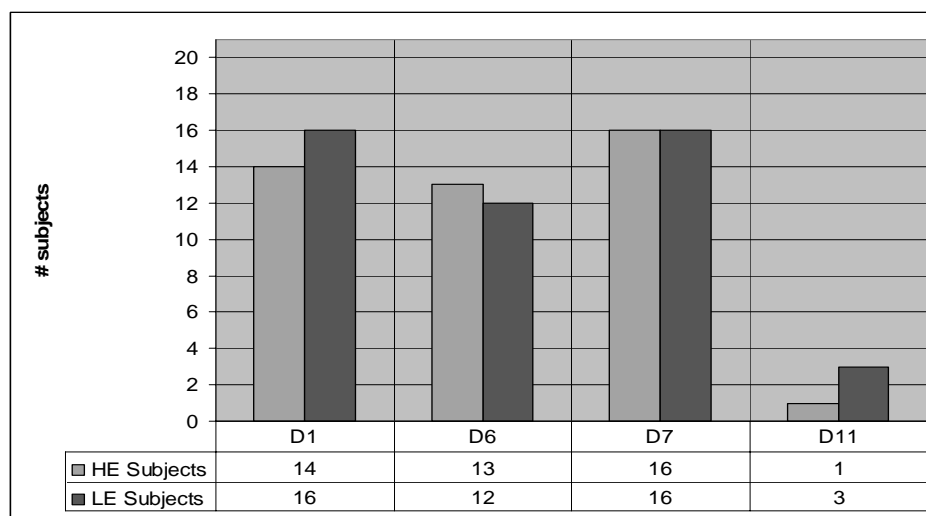


Figure 29 ‘Wrong statement’ defects detected by the subjects

The results of the t-Test on the ‘wrong code statement’ defects data are given below:

t-Test: Two-Sample Assuming Unequal Variances		
	Highly-experienced Subjects (HE)	Less-experienced subjects (LE)
Mean	2,10	2,19
Variance	0,79	1,36
Observations	21,00	21,00
Hypothesized Mean Difference	0,00	
Degree of Freedom (df)	37,00	
t Stat	-0,30	
P(T<=t) one-tail	0,38	
t Critical one-tail	1,69	

Table 16 t-Test conducted on the ‘wrong code statement’ defects

Since the first mean is smaller than the second one, and the resulting t-Stat is a negative number, -0.30, we can not reject the null hypothesis that the highly-experienced reviewers do not detect more ‘wrong code statement’ defects than the less-experienced reviewers. Even if we consider the opposite of our original hypothesis and consider that the less-experienced subjects detect more ‘wrong code statement’ defects than the highly-experienced reviewers, we do not get statistically significant results.

We will now have a closer look at the figure given below to check the defects that were poorly addressed by the subjects, regardless of their educational background and experience.

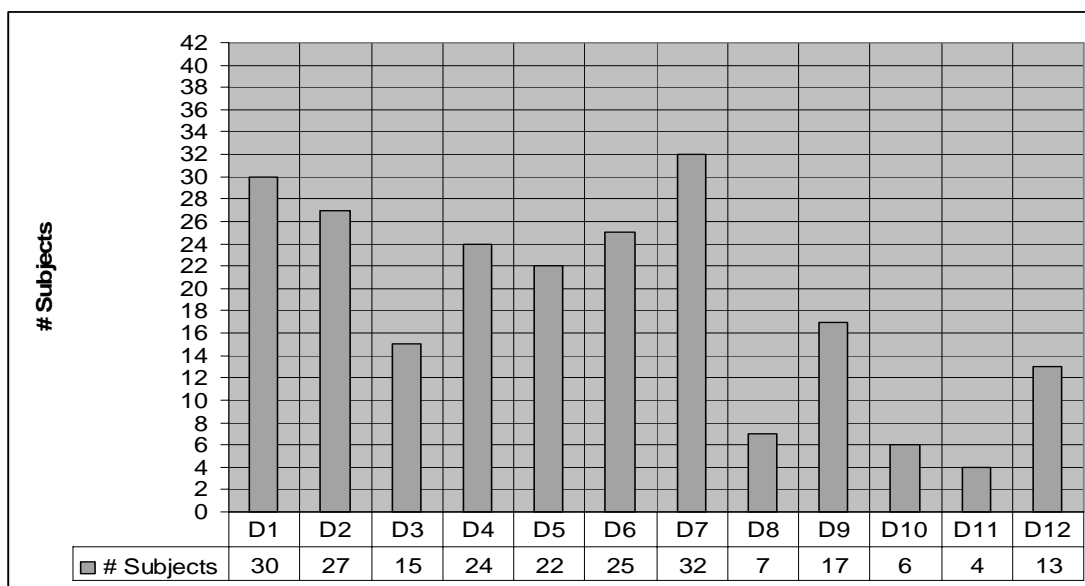


Figure 30 # subjects who successfully detected the defects

As shown in the figure, D8, D10, and D11 were found by the least number of the subjects (regardless of their experience). We believe that this factor is related to a shortcoming in the checklist-based method, which is related to the checklist’s length. This issue is further discussed in the next section.

7.2. Data Interpretation

In this section, we will try to interpret the data and attempt to answer the following questions:

- *What are the reasons for such a great variation in the effectiveness of the subjects having almost the same educational background and experience?*

The number of defects detected by the subjects having almost the same educational background and experience varied significantly, which we believe, could be correlated to the lack of motivation of the subjects. The number of detected defects varied from 1 to 9 for the highly-experienced subjects and 3 to 11, in case of the less-experienced subjects. The less-motivated subjects would read the source code only once and be content with that, regardless of the number of the defects detected, while the highly-motivated subjects would read the source code, or at least part of it, several time to find as many defects as possible.

Another reason for such behavior could be the students' scant attention in static reviews because they do not see the bigger picture in the relatively simple software inspection session and under-estimate its effectiveness in software validation.

The motivation of the subjects is hard to control and quantify. In our experiment, all the subjects were offered a modest inducement for their participation. All the subjects received the same inducement regardless of the number of the defects detected by them. We might have been able to partly control this variable, if the subjects had received an inducement amount according their effectiveness in defect detection.

Another option is to make such experiments, as a part of graded projects, where more effective subjects are given a better grade than the less efficient ones. These measures can help us getting the students to appreciate the motivation for expending maximum effort during the defect detection activity.

- *Why the HE subjects did not perform more effectively than the LE subjects?*

Overall, the HE subjects were not more effective than the LE subjects. The results from the first hypothesis showed that the software development experience and education did not help the subjects' performance. It should be noted that most of the subjects in the HE group were PhD students, while most of the subjects in the LE group came from third and fourth year. The PhD students worked with the issues at the software architectural level, while the LE subjects had recently taken advanced programming courses which involved many practical programming exercises.

Thus, one possible explanation for this result was that the LE subjects had more 'hands-on' programming experience than the HE subjects. This might have neutralized the effect of longer education and higher software development experience, which meant that spending about the same effort, the LE subjects found more defects.

As mentioned in the first chapter, the experienced reviewers are considered to be more effective than the inexperienced reviewers. However the research done in this area has indicated controversial and inconsistent results. Our results are consistent with [Basili01], who did not investigate the relationship between the reviewer's experience and the classes of defects detected by them; but the reviewer's overall effectiveness. Basili did not find any statistically significant improvement in the reviewer's effectiveness which could be correlated to their technical experience, domain knowledge, and software development experience.

Similarly, the studies done by [Laitenberger96] were unable to find any superiority for the experienced reviewers.

- *Is there any relationship between the reviewers' experience and classes of defects detected by them? What classes of defects are normally detected by the experienced reviewers? Are there any types of defects that are generally detected more easily by less-experienced reviewers than highly-experienced reviewers?*

Analysis of the second hypothesis showed statistically significant superiority of the LE subjects in detecting 'missing code statement' defects. The third hypothesis concerned 'extra code statement' defects, favored the HE subjects, while the fourth hypothesis, about the 'wrong code statement' defects did not give any statistically significant result. In case of the 'missing code statement' defects it seemed like the LE subjects read the source code line-by-line and focused on each line to check if there was anything missing.

On the other hand, the HE subjects focused on the overall functionality of the software and might not read all the lines of the source code. It seems as if the HE subjects did not just check for mistakes in the code sequentially but concentrated on their favorite aspects of the code depending on their expertise and experience e.g. D3 was a trivial defect; the main method was missing the keyword 'static'. Anyone who has programmed in Java can hardly miss that defect, unless he is not focusing on that defect at all.

We believe that the HE subjects focused more on the issues concerning the logic and maintenance issues than the LE subjects. Trivial 'missing code statement' mistakes were not even considered by them, while reading the code for defect detection. The experiment conducted by [Kelly97] showed that the experienced reviewers identified a higher proportion of findings that required deeper understanding of the code's overall functionality.

The hypothesis regarding 'extra code statement' defects supports this explanation, where the HE subjects were more effective. We believe that most of the LE subjects that have no experience with looking for maintenance issues are not used to thinking about the effect of extra statements on later stages of the life cycle. For example, a variable that has been declared but never used would not pose any danger to the program execution.

Due to this lack of experience, the LE subjects focus more on the statements that are necessary for the software to work properly (missing/wrong statements) than the extra statements, while the HE subjects focus on the maintenance issues (extra statements).

➤ *Why were some defects not properly addressed by the subjects of both groups?*

D8, D10, and D11 were the defects that were detected by the least number of the subjects in the both groups. We had the same problem during the experiment that we conducted last semester, where we could correlate those least found defects with the length of the checklist [Awan03]. The checklist used in the experiment had twelve sections with a total of 47 questions. Checklists used in the industry are often longer and contain more questions. It seems like most of the defects (especially D8 and D10) that were not properly addressed by the reviewers are from the very last sections of the checklist.

There could be two types of subjects; those who read the checklist sequentially and the other who pick sections according to their favorite issues. The subjects who did not read the checklist sequentially might just ignored the last sections because they considered them to be more complex than other questions in the checklist. The sequential readers might not have reached the very last sections, due to lack of time or motivation.

The limited inspection time (45 minutes) could be the cause that some inspectors were unable to complete the last sections of the checklist. We suggest conducting a replication of the experiment where the subjects are allowed to use as much time as they need to clarify these results.

D11, the defect that was detected by the least number of the subjects, was not properly addressed by the checklist. We believe that only the subjects that had recently worked with that kind of defect in the near past could identify D11, otherwise it was a difficult defect to be detected by static review. This amplifies the point that the quality of the checklist is one of the most important factors; bad checklists limit the creativity of the reviewers regardless of their experience.

➤ *How can these results be used in guiding the industry to reap the maximum benefits of inspections?*

The most important thing that our results have indicated is that the classes of detected defects differ according to the reviewer's educational background and experience. We believe that this information can guide a project leader to form an ideal reviewing including consisting of members with different experience and background. It is not always the optimal solution (in terms of cost and types of detected defects) to include only the most experienced reviewers in the inspection team. To ensure that all classes of defects are detected, the inspection teams must have a combination of novice and experienced reviewers.

In this manner, the inspection team would not only be able to find the hard-to-find defects but also the typical-novice-programmers mistakes that are equally dangerous and mostly overseen by the experienced reviewers.

Furthermore, we recommend a new inspection method that combines divide-and-conquer and round-robin strategies. The method is described in 'Recommendations' section in the next chapter.

8. Conclusions and Further Work

Success never comes by any magic formula, need three ingredients which when combine together can work wonder.

- Willingness to work hard*
 - Enthusiasm to succeed*
 - Potential supervision*
-

- **Section 8.1 ‘Summary’:** This section provides a short summary of the whole experiment.
- **Section 8.2 ‘Conclusions’:** The discussions about the findings and the conclusions are presented in this section.
- **Section 8.3 ‘Recommendations’:** Based on the conclusions, this section recommends a new inspection strategy.
- **Section 8.4 ‘Further Work’:** This section includes suggestions for the future work.

8.1. Summary

The major task of this study was to design and conduct an experiment that investigated the effects of the reviewer's background on the number and types of the defects detected by them. During our literature review, we found that there was no definite answer to the effects posed by these sources of variations. Therefore, the question of finding the most effective combination of defect detection technique and the above-mentioned sources of variations became the focus of our experiment.

The subjects included forty-two undergraduate, graduate and PhD students who were supposed to inspect 124 lines of Java code by using the checklist-based method. They were divided into two groups of equal size; the less-experienced subjects and the highly-experienced subjects. Unlike the highly-experienced subjects, the less-experienced subjects had less (or no) experience with industrial software projects and most of them had never participated in a formal software inspection. Most of the highly-experienced subjects were PhD students with industrial experience.

The experiment had one factor with two treatments, since we had two student profiles depending on their experience. All subjects were required to inspect the same client-server Java application. The inspected source was a part of a Java client-server application in which the multi-threaded server encrypted the text given by the client and returned it to the client. The subjects inspected the code individually; there was no group activity involved.

The defect classification used in the experiment was a simple subset of the Orthogonal Defect Classification Scheme [Link6]. We divided the defects into three classes; wrong statements, extra statements, and missing statements. Twelve defects were seeded into the source code and each defect class was represented by four seeded defects. Our hypotheses were that the experienced subjects would be more effective than the less-experienced subjects in detecting defects from all the defect classes, but the results revealed some other patterns. The table given below shows the summary of the hypothesis-testing results:

	HE subjects	LE subjects	Statistically significant result
Missing statements	16.75%	45.25%	LE > HE
Extra statements	58.25%	35.75%	HE > LE
Wrong statements	52.50%	54.75%	None
Overall	42.42%	45.58%	None

Table 17 Summary of the hypothesis-testing results (% of defects detected)

Overall, the HE subjects detected 42.42% of true defects on average, while the LE subjects detected 45.58%. The LE subjects were slightly more effective than the HE subjects in detecting true defects. The results did not show superiority of the HE subjects to the LE subjects. The patterns revealed in the data analysis indicated a relationship between the reviewers' experience and the type of the defects detected by them. We found statistically significant superiority of the LE subjects in detecting the 'missing code statement' defects. In case of the 'extra code statement' defects, the HE subjects were significantly more effective than the LE subjects. The results from the 'wrong code statement' defects did not give any statistically significant result.

We suggested a new inspection method that can combine divide-and-conquer and round-robin strategies. These strategies can be used to divide sections of the checklist among the reviewers. This could solve the problem of limited inspection time and ensure that all the sections of the checklist are properly addressed.

We tried to remove the validity threats or at least be aware of the limitations caused by their presence, which in turn helped us to make informed decisions about the best way to reduce their effects. We had limited resources available for the experiment but the ideas that we have gained through this exploration have provided a valuable input for further work in this area.

8.2. Conclusions

It should be kept in mind that a single study like this one is no sufficient basis for changing the attitudes concerning the questions of defect detection techniques and other sources of variations. However, the results given below can provide project leaders with important clues when selecting the members for an inspection team. Furthermore, the results can give us an opportunity of coming up with new insights, which will lead to new research.

Based on the above-mentioned observations we can draw the following conclusions:

- The reviewers that have longer education and more experience are not necessarily more effective than the less-experienced reviewers; hands-on programming experience is more important in determining the reviewers' effectiveness.
- There is a relationship between the reviewers' experience and classes of defects the detected. The HE subjects focus more on the issues concerning the logic and maintenance issues than the LE subjects and trivial 'missing code statement' mistakes can thus be overseen. On the other hand, most of the LE subjects that have no experience with looking for maintenance issues are less effective in detecting the defects related to those issues, which can include 'extra code statement' defects.
- Quality of the checklist is a critical factor for successful reviews and bad checklists can limit the creativity of the reviewers regardless of their experience. Some reviewers tend to ignore the complex sections of the checklist, which may be due to limited inspection time or lack of motivation. However, deciding the optimal size of the checklist is not a trivial task. If it is short, e.g., one page, then it cannot address all the important classes of defects and the questions contained in it would not be specific. This can decrease the effectiveness of less-experienced reviewers. If the length is more than one page, some reviewers might not be able to check the whole list. This conclusion is consistent with the results that we found during the last semester.
- Motivation of the subjects is one of the most important factors in determining their effectiveness. The number and type of the defects detected by the subjects having almost the same educational background and experience can vary significantly, which we believe, could be due to their lack of motivation.

- A good inspection team should include both novice and experienced reviewers to ensure that the inspection team would not only be able to find the hard-to-find defects but also the typical-novice-programmers mistakes that are equally dangerous and mostly overseen by the experienced reviewers.

8.3. Recommendations

Some sections of the checklist are generally ignored by the reviewers, either due to lack of time or motivation. To cope with these difficulties, we suggest a new inspection method that is a combination of divide-and-conquer and round-robin strategies.

First we can use the divide-and-conquer strategy and divide different sections of the checklist among the reviewers. In this manner, each reviewer would have enough time to go through all the questions in his sections, which would ensure that all the sections of the checklist are properly addressed. This approach would give a sense of responsibility to the reviewers about the issues that were addressed by their sections.

If, for some reason, a ‘to-be-held responsible’ situation is not desirable, the reviewers can be requested to exchange sections between them in a round-robin fashion. This would ensure that each section is reviewed by multiple reviewers having different backgrounds, where each reviewer would go through his section and read the code according to his experience and interests. By using the round-robin strategy, each reviewer would provide ‘a new pairs of eyes’ which would ensure better coverage of the source code.

Furthermore, we recommend conducting an experiment which will investigate the motivation factor of the subjects. The experiment can involve three groups of subjects, where one group can be given inducement according to their performance and the other group would be offered same inducement regardless of their effectiveness. The last group can have the inspection session as a part of their graded coursework, where the grade is given according to their effectiveness. We believe that the motivation level would differ significantly among these three groups and might provide us with some potential research ideas.

8.4. Further Work

Researchers do not take even their own observations seriously or accept them as scientific observations until they have repeated and tested them [Popper60]. We obtained the results from a single experiment, with limited resources available. Because of this, we suggest that other researcher would attempt to replicate our experiment. Conducting the same analysis on data from existing experiments as well as new replications will bring more clarity into the advantages and disadvantages of the checklist-based reading and its optimal combination with other sources of variations.

We have prepared all the necessary material to facilitate this. We have rigorously defined our experiment and have tried to control the most important threats to validity, but still replications can significantly increase our confidence that these threats was adequately addressed.

We suggest planning an experiment that will use the combination of divide-and-conquer and round-robin strategies, as suggested in section 8.3. Such an experiment can investigate the key factors like length of the checklist, its complexity and structure and its relation with reviewers experience as well as with specific classes of defects detected by them. The experiment can select the subjects according to the guidelines given in the previous section and study the subject's motivation factor in detail.

The study can try to answer several questions. Why do some inspectors ignore specific classes of defects? Is it merely related to complexity and structure of the checklist or inspector's experience or some other factor? What is the optimal size of the checklist? Should it be same for all reviewers regardless of their experience and expertise of a specific domain? Can we use a combination of several techniques?

Many studies have investigated issues concerning technical factors and their influence of effectiveness and efficiency on the inspection process; however, the effects of social factors on the inspection process have recently caught our attention. More research is needed in this area. Since most of the social factors depend heavily on psychological factors, a close collaboration with other social sciences departments would be appropriate for further work.

Further work with this experiment can also help to investigate how much variation in the observed defect detection effectiveness can be explained by natural variation factors outside our control e.g. inspector's experience, quality of the source code etc. Only 25% of the variation in inspection process can be accounted for by process inputs such as documents and inspectors [Votta98]. The rest is hidden somewhere within the inspection process and has not been accounted for yet.

8. References

- [Ackerman83] A.F. Ackerman, P.J. Fowler and R.G. Ebenau, *Software inspections and the industrial production of software*, in Hans-Ludwig Hausen (Ed.), *Software validation: Inspection, testing, verification, alternatives*, Proceedings of the Symposium on Software Validation, pp. 13–40, Darmstadt, FRG, 1983.
- [Armbrust02] Ove Armbrust, *Developing a Characterization Scheme for Inspection Experiments*, Master's Thesis at Universit'at Kaiserslautern, 2002.
- [Awan03] Tanveer Hussain Awan. *Sources of variations in software inspections: An empirical and explorative study*. TDT4735 Software Engineering, NTNU Norway, 2003.
- [Ballman94] K. Ballman and L. G. Votta. *Organizational congestion in large-scale software development*. Proceeding of the Third International Conference on the Software Process. Applying the Software Process, IEEE CS Press: 123-34, 1994
- [Barnard94] J. Barnard and A. Price, *Managing Code Inspection Information*. IEEE Software, 11(2):59-69, 1994.
- [Basili87] Basili, V.R., and Selby, R.W. "Comparing the Effectiveness of Software Testing Strategies." *IEEE Transactions on Software Engineering*, SE-12 (7): 1278-1296, Dec. 1987.
- [Basili90] Rombach, H.D. and V.R. Basili. 'Practical benefits of goal-oriented measurement', in Proc. Annual Workshop of the Centre for Software Reliability: Reliability and Measurement. Garmisch-Partenkirchen, Germany: Elsevier, 1990.
- [Basili96] Victor R. Basili, Scott Green, Oliver Laitenberger, Filippo Lanubile, Forrest Shull, Sivert Sørumgård, Marvin V. Zelkowitz. *The Empirical Investigation of Perspective-Based Reading*. 1996.
- [Basili01] Victor Basili, Jeffrey Carver, Forrest Shull. *Investigating the effect of Process Experience on Inspection Effectiveness*. University of Maryland Institute for Advanced Computer Studies, 2001.
- [Bisant89] David B. Bisant and James R. Lyle. *A Two-Person Inspection Method to Improve Programming Productivity*, IEEE Transactions on Software Engineering, Volume 15, Number 10, pages 1294-1304, October 1989.
- [Boehm87] B. W. Boehm, *Improving Software Productivity*, Computer, Vol. 20, No. 9, pp. 43-47, 1987.

- [Chernak96] Y. Chernak, *A Statistical Approach to the Inspection Checklist Formal Synthesis and Improvement*. IEEE Transactions on Software Engineering, 22(12):866–874, 1996.
- [Collofello89] J. S. Collofello and S. N. Woodfield, *Evaluating the effectiveness of reliability-assurance techniques*. Journal of Systems and Software, 9:191-195, 1989.
- [Conradi99] R. Conradi, A. S. Marjara and B. Skatevik, *Empirical Studies of Inspection and Test Data*, in Proceedings of the First Conference on Product-Focused Process Improvement, Oulo, Finland, 1999.
- [Cook79] Cook, T. D. & Campbell, D. T. *Quasi-experimentation: Design and analysis issues for field settings*. Boston: Houghton Mifflin, 1979.
- [Emam98] Khaled El Emam and Isabella Wieczorek. *The Repeatability of Code Defect Classifications*. Fraunhofer Institute for Experimental Software Engineering, 1998.
- [Fagan76] M. E. Fagan. *Design and code inspections to reduce errors in program development*. IBM Systems Journal, 15(3):182–211, 1976.
- [Fagan86] M. E. Fagan. *Advances in Software Inspections*. IEEE Transactions on Software Engineering, 12(7):744-751, 1986.
- [Gilb93] T. Gilb, and D. Graham, *Software Inspection*. Addison-Wesley 1993.
- [Grady94] R. B. Grady and T. Van Slack, *Key Lessons in Achieving Widespread Inspection Use*. IEEE Software, 11(4):46-57, 1994.
- [Hallum01] Arne Marius Hallum. *Design patterns and maintenance: finding the relationship*. IDI, NTNU, 2001.
- [Hollocker87] C.P. Hollocker, *The standardization of software reviews and audits*, in G.G. Schulmeyer & J.I. McManus (Eds.), *Handbook of Software Quality Assurance*, pp. 211–266, Van Nostrand-Reinhold Company, NY, 1987.
- [IEEE89] IEEE. *IEEE Standard for Software Reviews*. IEEE Press, 1989
- [IEEE97] IEEE. *IEEE Standard for Software Reviews*. IEEE Press, 1997
- [Kallakuri00] Praveen Kallakuri, Sebastian Elbaum. *Experimental Studies in Empirical Software Engineering* [online]. Available: <http://www.acm.org/crossroads/xrds7-4/empirical.html>, 2001
- [Kan95] S. H. Kan, *Metrics and Models in Software Quality Engineering*. Addison-Wesley Publishing Company, 1995.

- [Kaner98] C. Kaner, *The Performance of the N-Fold Requirement Inspection Method*, Requirements Engineering Journal, vol. 2, no. 2, pp. 114-116, 1998.
- [Kelly92] J. C. Kelly, J. S. Sherif and J. Hops, *An Analysis of Defect Densities found during Software Inspections*. Journal of Systems and Software, 17:111-117, 1992.
- [Kelly97] Diane Kelly and Terry Shepard. *Task-Directed Software Inspection Technique: An Experiment and Case Study*. Royal Military College Of Canada, 1997.
- [Knight93] John C. Knight and Ethella Ann Myers. *An Improved Inspection Technique*, Communications of the ACM, Volume 11, Number 11, pages 51-61, 1993.
- [Johnson93] Philip M. Johnson and Danu Tjahjono. *CSRS users guide*. Technical Report ICS-TR-93-16, Collaborative Software Development Laboratory, Department of Information and and Computer Sciences, University of Hawaii, 1993.
- [Johnson94] Philip M. Johnson. *An instrumented approach to improving software quality through formal technical review*. In Proceedings of the 16th International Conference on Software Engineering, pages 113--122, Sorrento, Italy, 1994.
- [Jude91] C. M. Judd, E. R. Smith, and L. H. Kidder. *Research Methods in Social Relations*. Holt, Rinehart and Winston, sixth edition, 1991.
- [Juristo01] Natalia Juristo, and Ana Moreno. *Basics of software engineering experimentation*. Boston : Kluwer Academic Publishers, 2001.
- [Land97] Lesley Pek Wee Land, Ross Jeffery, and Chris Sauer. *Validating the Defect Detection Performance Advantage of Group Designs for Software Reviews: Report of a Replicated Experiment*, Australian Software Engineering Conference, ASWEC 1997.
- [Laitenberger98] Oliver Laitenberger. *Studying the Effects of Code Inspection and Structural Testing on Software Quality*. Fraunhofer Institute for Experimental Software Engineering. ISERN-Report ISERN-98-10, 1998.
- [Laitenberger99] Oliver Laitenberger, Colin Atkinson, Maud Schlich, and Khaled El-Emam . *An Experimental Comparison of Reading Techniques for Defect Detection in UML Design Documents*. National Research Council Canada, 1999.

- [Laitenberger00] Oliver Laitenberger, C. Atkinson, M. Schlich, K. Emam, *An Experimental Comparison of Reading Techniques for Defect Detection in UML Design Documents*. Journal of Systems and Software, 53:2, 183-204, 2000.
- [Laitenberger02] Oliver Laitenberger, *A Survey of Software Inspection Technologies*, in Handbook on Software Engineering and Knowledge Engineering*, Vol. II, World Scientific Publishing, 2002.
- [Madachy93] R. Madachy, L. Little, and S. Fan, *Analysis of a successful Inspection Program*. Proceeding of the 18th Annual NASA Software Eng. Laboratory Workshop, pages 176-198, 1993
- [McConnell93] S. McConnell, *Code Complete*. Microsoft Press, 1993.
- [McGibbon96] T. McGibbon, *A Business Case for Software Process Improvement*. Technical Report F30602-92-C-0158, Data & Analysis Center for Software (DACS).URL: <http://www.dacs.com/techs/roi.soar/soar.html>, 1996.
- [Mills79] H. D. Mills, Richard C. Linger and Bernard I. Witt. *Structured Programming: Theory and Practice*. Addison-Wesley Publishing Company, 1979.
- [Mills82] H. D. Mills, V. R. Basili. *Understanding and Documenting Programs*. IEEE Transactions on Software Engineering, SE-8(3):pp 270-283, May 1982.
- [Miller98] J. Miller, M. Wood and M. Roper. *Further Experiences with Scenarios and Checklists*, Empirical Software Engineering, Volume 3, 37 – 64, 1998.
- [Myers78] Glenford J. Myers. *Experiment in Program Testing and Code Walkthroughs/Inspections*, IBM Systems Research Institute, 1978.
- [Perry00] Perry, E.D., Adam A.P, And Lawrence G.V. *Empirical Studies of Software Engineering: A Roadmap*. Future of Software Engineering, Limerick, Ireland, 2000.
- [Parnas85] David L. Parnas , David M. Weiss, *Active design reviews: principles and practices*, Proceedings of the 8th international conference on Software engineering, p.132-136, August 28-30, 1985
- [Parnas87] D. Parnas, and D. Weiss, *Active Design Reviews: Principles and Practice*. Journal of Systems and Software, 7:259–265, 1987.
- [Porter94] A. A. Porter and L. G. Votta. *An experiment to assess different defect detection methods for software requirements inspections*. NASA Research Paper, 1994.

- [Porter97] A. A. Porter and P. M. Johnson. *Assessing software review meetings: Results of a comparative analysis of two experimental studies*. *IEEE Transactions on Software Engineering*, 23(3):129–144, 1997.
- [Popper60] Kr. Popper. *The logic of Scientific Discovery*. London: Hutchinson, 1960.
- [Remus84] H. Remus, *Integrated Software Validation in the View of Inspections/ Reviews*. *Software Validation*, pages 57-65, 1984.
- [Robson93] C. Robson. *Real World Research*, Blackwell, UK, 1993.
- [Russel91] G. W. Russel. *Experience with inspection in ultralarge-scale developments*. *IEEE Software* 8(1): 25-31, 1991
- [Schnieder92] G. M. Schneider, J. Martin, W Tsai. *An Experimental Study of Fault Detection in User Requirements Documents*. *ACM Trans. on Software Engineering and Methodology*, 1(2): 188-204, 1992.
- [Sommerville92] I. Sommerville, *Software Engineering*, Fourth Edition, AddisonWesley, Wokingham, 1992.
- [Thelin03] Thomas Thelin and Per Runeson. *An Experimental Comparison of Usage-Based and Checklist-Based Reading*. TSE'03 - *IEEE Transactions on Software Engineering*, 29(8):687-704, 2003.
- [Tripp91] L. L. Tripp, W. F. Stuck, and B. K. Pflug, *The Application of Multiple Team Inspections on a Safety-Critical Software Standard*. *Proceedings of the fourth Software Engineering Standards Application Workshop*, pages 106-111, 1991.
- [Trochim99] W. M. Trochim. *Research methods knowledge base*. Available online: <http://trochim.cornell.edu/kb/>, 1999.
- [Votta93] L. G. Votta, *Does Every Inspection Need a Meeting?* In *Proceedings of ACM SIGSOFT '93*, pp 107-114. ACM, Dec. 1993.
- [Votta97] L. G. Votta, H. P. Siy, C. A. Toman, and A. A. Porter. *An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development*. *IEEE Transactions on Software Engineering*, 23(6):329-346, 1997
- [Votta98] L. G. Votta, Adam A. Porter; Harvey P Siy. *Understanding the Sources of Variation in Software Inspections*. In: *ACM Transactions on Software Engineering and Methodology* 7 (1998), January, Nr. 1, 41–79, 1998
- [Weller92] E. F. Weller. *Experiences with Inspections at Bull HN Information System*. *Proceedings of the fourth Annual Software Quality Workshop*, 1992.

- [Weller93] E. F. Weller, *Lessons from Three Years of Inspection Data*. IEEE Software, 10(5):38-45, 1993.
- [Wohlin00] Claes Wohlin. *Experimentation in Software Engineering: An Introduction (The Kluwer International Series in Software Engineering)*, 2000.
- [WohlinRegnell00] Claes Wohlin, Martin H, Björn Regnell. *Using Students As Subjects - A Comparative Study of Students and Professionals in Lead-Time Impact Assessment*. Empirical Software Engineering 5, 201-214, 2000.
- [Link1] <http://www.softwareqatest.com/qatfaq1.html>
- [Link2] <http://www.construx.com/survivalguide/glossary.htm#W>
- [Link3] http://wwi.cs.unimagdeburg.de/lehre/ss2000/swdev/skript/gdpa_d/methods/ma-rev.htm
- [Link4] <http://www.tvu.ac.uk/dissguide/hm1u1/hm1u1text2.htm>
- [Link5] http://www.iteva.rug.nl/gqm/templates/GQM_part2.ppt
- [Link6] <http://www.research.ibm.com/softeng/ODC/ODC.HTM>

Appendix A: Definitions

Defect: A manifestation of an error in hardware/software. A fault, if encountered, may cause a failure.

Error: Human action that results in software containing a fault, e.g. omission or misinterpretation of user requirements.

Failure: The inability of a system or a system component to perform a required function within specified limits. A failure may be produced when a fault is encountered.

Fault See *defect*.

Validation To determine whether the right product is being made, i.e. do the requirements reflect the user needs?

Verification To determine whether the product is being made correctly, i.e. are the requirements being implemented correctly?

Appendix B: Source code (Without defects)

B.1. EncryptionClient.java

```
import java.io.*;
import java.net.*;

/**
 * Sends the user's input to the EncryptionServer, which returns the encrypted version
 * of the input.
 */
public class EncryptionClient {
    public static void main(String[] args) throws IOException {
        // connection to the server
        Socket socket = null;
        // write to the server
        PrintWriter out = null;
        // read from the server
        BufferedReader in = null;
        // text sent from the server
        String fromServer;
        // text given from the user via command line
        String fromUser;

        try {
            // open up the connection on the given port
            socket = new Socket("localhost", 4444);
            // prepare input/output streams to the server
            out = new PrintWriter(socket.getOutputStream(), true);
            in = new BufferedReader(new
                InputStreamReader(socket.getInputStream()));
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host: encryptionHost");
            System.exit(1);
        }

        // get input from the user via command line, until the server sends "Bye"
        BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
        while ((fromServer = in.readLine()) != null) {
            System.out.println("Server: " + fromServer);
            if (fromServer.equals("Bye.")) break;
            fromUser = stdIn.readLine();
            if (fromUser != null) {
                System.out.println("Client: " + fromUser);
                out.println(fromUser);
            }
        }
    }
}
```

```
        // cleanup
        out.close();
        in.close();
        stdIn.close();
        socket.close();
    }
}
```

B.2. EncryptionServer.java

```
import java.net.*;
import java.io.*;

/**
 * EncryptionServer loops forever, listening for client connection requests
 * on a ServerSocket. When a request comes in, EncryptionServer accepts the
 * connection, creates a new EncryptionServerThread object to process it,
 * hands it the socket returned from accept, and starts the thread. Then
 * the server goes back to listening for connection requests. The
 * EncryptionServerThread object communicates to the client by reading from
 * and writing to the socket
 */
public class EncryptionServer {
    public static void main(String[] args) throws IOException {
        // connection socket
        ServerSocket serverSocket = null;
        // keep running the server
        boolean listening = true;

        try {
            // open the socket
            System.out.println("Up and running");
            serverSocket = new ServerSocket(4444);
        } catch (IOException e) {
            System.err.println("Could not listen on port: 4444.");
            System.exit(-1);
        }
        // keep running the server
        while (listening) new EncryptionServerThread(serverSocket.accept()).start();
        serverSocket.close();
    }
}
```

B.3. EncryptionServerThread.java

```
// separate thread for each client
class EncryptionServerThread extends Thread {
    // connection to the client
    private Socket socket = null;

    // constructor
    public EncryptionServerThread(Socket socket) {
        this.socket = socket;
    }

    // keep talking to the client, until the SecretEncryptor sends "Bye" to us
    public void run() {
        System.out.println("Got a new client");

        try {
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(
                new InputStreamReader(socket.getInputStream()));
            String inputLine, outputLine;

            out.println("Welcome to the encryption server");
            while ((inputLine = in.readLine()) != null) {
                outputLine = "output>>" + SecretEncryptor.encrypt(inputLine);
                out.println(outputLine);
                if (outputLine.equals("Bye")) break;
            }
            out.close();
            in.close();
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Appendix C: Source code (With defects)

C.1. EncryptionClient.java

```
import java.io.*;
import java.net.*;

/**
 * Sends the user's input to the EncryptionServer, which returns the encrypted version
 * of the input.
 */
public class EncryptionClient {

    // the main method for this class
    public void main(String[] args) throws IOException {
        boolean validInput = false;
        // connection to the server
        Socket socket = null;
        // write to the server
        PrintWriter out = null;
        // read from the server
        BufferedReader in = null;
        // text sent from the server
        String fromServer;
        // text given from the user via command line
        String fromUser;

        try {
            // open up the connection on the given port 4445
            socket = new Socket("localhost", 4444);
            // prepare input/output streams to the server
            out = new PrintWriter(socket.getOutputStream(), true);
            in = new BufferedReader(new
                InputStreamReader(socket.getInputStream()));
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host: encryptionHost");
            System.exit(1);
        }

        // get input from the user via command line, until the server sends "Bye"
        BufferedReader stdIn = new BufferedReader(new
            InputStreamReader(System.in));
        while ((fromServer = in.readLine()) != null) {
            System.out.println("Server: " + fromServer);
            if (fromServer.equals("Bye.")) break;
            fromUser = stdIn.readLine();
        }
    }
}
```

```
        if (fromUser != null) {
            System.out.println("Client: " + fromUser);
            out.println(fromUser);
        }
    }
    // cleanup
    // out.close();
    in.close();
    stdIn.close();
    socket.close();
}
}
```

C.2. EncryptionServer.java

```
import java.net.*;
import java.io.*;

/**
 * EncryptionServer loops forever, listening for client connection requests
 * on a ServerSocket. When a request comes in, EncryptionServer accepts the
 * connection, creates a new EncryptionServerThread object to process it,
 * hands it the socket returned from accept, and starts the thread. Then
 * the server goes back to listening for connection requests. The
 * EncryptionServerThread object communicates to the client by reading from
 * and writing to the socket
 */
public class EncryptionServer {
    public static void main(String[] args) throws IOException {
        // connection socket
        ServerSocket serverSocket = null;
        // keep running the server
        boolean listening = false;

        try {
            // open the socket
            serverSocket = new ServerSocket(4444);
        } catch (IOException e) {
            System.err.println("Could not listen on port: 4444.");
            System.exit(-1);
        }
        // keep running the server
        while (listening) {
            new EncryptionServerThread(serverSocket.accept()).start();
        }
        serverSocket.close();
    }
}
```

C.3. EncryptionServerThread.java

```
// separate thread for each client
class EncryptionServerThread extends Thread {
    // connection to the client
    private Socket socket = null;

    // constructor
    public EncryptionServerThread(Socket socket) {
        this.socket = socket;
    }

    // keep talking to the client, until the SecretEncryptor sends "Bye" to us
    public void run() {
        System.out.println("Got a new client");

        try {
            String[] messages = new String[100];
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(
                new InputStreamReader(socket.getInputStream()));
            String inputLine, outputLine;
            while ((inputLine = in.readLine()) != null) {
                outputLine = "output>>" + SecretEncryptor.encrypt(inputLine);
                int length = outputLine.length();
                out.println(outputLine);
                if (outputLine.equals("Good Bye")) break;
            }
            out.close();
            in.close();
            socket.close();
        } catch (IOException e) {
            log(e.getMessage());
        }
    }

    // Appends the given message into the log file
    private static void log(String mes) {
        try {
            File logFile = new File("log.txt");
            FileWriter out = new FileWriter("log.txt", false);
            out.write(mes);
        } catch (IOException e) {
        }
    }
}
```

Appendix D: Java Inspection Checklist

1. Variable, Attribute, and Constant Declaration Defects (VC)

1. Are descriptive variable and constant names used in accord with naming conventions?
2. Are there variables or attributes with confusingly similar names?
3. Is every variable and attribute properly initialized?
4. Could any non-local variables be made local?
5. Are all for-loop control variables declared in the loop header?
6. Are there variables or attributes that should be constants?
7. Do all attributes have appropriate access modifiers (private, protected, public)?
8. Are there static attributes that should be non-static or vice-versa?

2. Method Definition Defects (FD)

1. Are descriptive method names used in accord with naming conventions?
2. Is every method parameter value checked before being used?
3. For every method: Does it return the correct value at every method return point?
4. Do all methods have appropriate access modifiers (private, protected, public)?
5. Are there static methods that should be non-static or vice-versa?

3. Class Definition Defects (CD)

1. Does each class have appropriate constructors and destructors?
2. Do any subclasses have common members that should be in the super-class?
3. Can the class inheritance hierarchy be simplified?

4. Data Reference Defects (DR)

1. For every array reference: Is each subscript value within the defined bounds?
2. For every object or array reference: Is the value certain to be non-null?

5. Computation/Numeric Defects (CN)

1. Are there any computations with mixed data types?
2. Is overflow or underflow possible during a computation?
3. For each expression with more than one operator: Are the assumptions about order of evaluation and precedence correct?

6. Comparison/Relational Defects (CR)

1. For every boolean test: Is the correct condition checked?
2. Are the comparison operators correct?
3. Has an "&" inadvertently been interchanged with a "&&" or a "|" for a "||"?

7. Control Flow Defects (CF)

1. Will all loops terminate?
2. When there are multiple exits from a loop, is each exit necessary and handled properly?
3. Does each switch statement have a default case?
4. Are missing switch case break statements correct and marked with a comment?
5. Do named break statements send control to the right place?
6. Can any nested if statements be converted into a switch statement?
7. Are null-bodied control structures correct and marked with braces or comments?
8. Does every method terminate?

8. Module Interface Defects (MI)

1. Are the number, order, types, and values of parameters in every method call in agreement with the called method's declaration?
2. If an object or array is passed, does it get changed, and changed correctly by the called method?

9. Comment Defects (CM)

1. Does every method, class, and file have an appropriate header comment?
2. Does every attribute, variable, and constant declaration have a comment?
3. Do the comments and code agree?
4. Are there enough comments in the code?
5. Are there too many comments in the code?

10. Input-Output Defects (IO)

1. Have all files been opened before use?
2. Are the attributes of the input object consistent with the use of the file?
3. Have all files been closed after use?
4. Are there spelling or grammatical errors in any text printed or displayed?

11. Layout and Packaging Defects (LP)

1. Is a standard indentation and layout format used consistently?
2. For each method: Is it no more than about 60 lines long?

12. Exceptions Handling (EH)

1. Are all relevant exceptions caught?
2. Is the appropriate action taken for each catch block?

Appendix E: Experience Form

Name:	
# semesters studied	
# year - Commercial experience in programming (any language)	
# year - Commercial experience in programming (Java)	
# programming courses taken e.g. programming, aldat etc	
Have you participated in any software inspection before? (Yes/No)	
In how many software development projects (with at least three members in the team) have you participated?	

Appendix F: Seeded Defects

Client

1. Wrong statement, line 26, Port number (4445 vs. 4444)
2. Extra statement, line 12, variables validInput not used
3. Missing statement, line 11, main method is missing the keyword static
4. Missing statement, line 46, out stream should have been closed

Server

5. Extra statement, line 49, array messages is never used
6. Wrong statement, line 58, “Good bye” instead of “Bye”
7. Wrong statement, line 19, Boolean listening should have been true
8. Missing statement, line 76, no exception handling
9. Extra statement, line 57, don’t need the variable length
10. Missing statement, line 75, nothing would be written to the file unless it is closed
11. Wrong statement, line 73, for appending the constructor parameter should have been true, not false
12. Extra statement, do not need line 72

Wrong statement defects: 4

Extra statement defects: 4

Missing statement defects: 4

Appendix G: Raw Data

Defects detected by the more experienced subjects. 1 = true (detected), Empty cell = false (not detected). X-axis = Defects ID, Y-axis = Subjects ID.

	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12
H1							1					
H2	1			1		1	1					1
H3	1	1			1		1		1			1
H4	1											
H5	1	1		1	1	1	1		1			
H6		1	1			1	1					
H7	1			1		1				1		
H8	1	1	1		1	1	1		1			
H9		1			1	1	1		1			
H10	1	1		1	1	1	1				1	1
H11	1	1			1		1					1
H12		1	1		1	1	1		1			1
H13	1	1			1	1	1	1	1	1		1
H14		1		1	1				1			1
H15	1	1				1						1
H16	1						1		1			
H17	1			1			1					
H18	1	1		1					1			1
H19		1			1	1	1	1	1			1
H20	1	1			1	1	1		1			
H21		1			1	1	1		1			

Defects detected by the less-experienced subjects. 1 = true (detected), Empty cell = false (not detected). X-axis = Defects ID, Y-axis = Subjects ID.

	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12
L1	1		1				1		1			
L2			1	1	1	1	1				1	
L3	1		1		1				1			1
L4			1	1	1							
L5		1			1	1	1					
L6		1	1	1			1	1				
L7	1			1		1		1				
L8	1	1		1		1	1			1	1	
L9	1	1	1	1		1	1					
L10	1	1		1		1	1		1			
L11	1			1			1	1	1			
L12	1	1		1	1	1	1					
L13	1			1		1	1					
L14	1		1		1							
L15			1	1	1							
L16	1	1		1			1	1				
L17	1	1		1		1	1					1
L18	1	1	1	1		1	1			1		
L19	1	1	1	1	1	1	1	1	1	1		1
L20	1	1	1	1	1	1	1			1	1	
L21	1	1	1	1	1		1					