AUTOMATED TOOL FOR SOFTWARE REQUIREMENTS INSPECTION

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Pradeep Amaran

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

April 2011

Fargo, North Dakota

North Dakota State University
Graduate School

Title

AUTOMATED TOOL FOR SOFTWARE

REQUIREMENTS INSPECTION

By

PRADEEP AMARAN

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

ABSTRACT

Amaran, Pradeep, M.S., Department of Computer Science, College of Science and Mathematics, North Dakota State University, April 2011. Automated Tool for Software Requirements Inspection. Major Professor: Dr. Gursimran S. Walia.

The software inspection process is a very cost effective method of identifying defects in documents produced during the software life cycle, leading to higher quality software with lower field failures. Manual inspections are labor intensive and dependent on human factors (e.g., preparation, moderation, and cooperation among development and quality assurance teams). An automated software inspection tool replaces a labor intensive manual approach of performing the inspection process. An automated inspection tool will offer greater efficiencies than any techniques involving manual inspections. Automation allows stakeholders (e.g., authors, inspectors) to closely work in coordination using the tool. Authors can host documents, view comments posted by inspectors, assign users and delete them. Inspectors can participate in the inspection process by validating against a set of guidelines and detect faults in a specific frame of time using different fault and error based inspection techniques. It is human to err, and as a result some of the faults may be overlooked. Hence, provisions are made for iterative inspection cycles to maximize the number of defects found and minimize the number of overlooked ones.

ACKNOWLEDGEMENTS

I would like to thank my major adviser, Dr. Gursimran S. Walia for his continued support, help and direction. I would also like to convey my gratitude to Dr. Kendall E. Nygard, Dr. Dean Knudson, and Dr. Marcelo J. Carena for being on my graduate committee. I would also like to thank my family and friends who encouraged me to complete my paper.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

In this competitive world it is essential to identify and eliminate defects from software and its artifacts and be able to develop software on time and with good quality. It is commonly understood that the majority of defects found in the early stages of software development via software inspection will improve the quality of the product while being cost effective. To address this problem, many approaches have been developed and evaluated through controlled case studies (e.g. [1, 2, 3, 4]). Considerable effort has been devoted to identifying methods to find and repair problems early in the software lifecycle when these repairs are easiest and cheapest. The goal of these methods is to detect and remove early-lifecycle faults i.e., mistakes recorded in a requirements or design artifact and code.

The use of software code inspections, design inspections, and requirements inspections has been found to increase software quality and lower software development costs [5, 6]. Prior studies indicate that inspections can detect as much as 93% of the total number of defects in an artifact [7]. Based upon a literature survey, on average, software inspections find 57% of the defects in code and design documents [8].

However, even when faithfully applying various empirically-validated fault-based techniques, software quality is still not at the desired level. It is estimated that 40-50% of the total project effort is spent on avoidable rework fixing problems that should have been fixed earlier in the lifecycle, or should have been prevented. Much of this rework is the result of the fact that early lifecycle fault detection techniques are based on incomplete fault taxonomies and do not lead developers to find all types of problems. Therefore, there

is still room for significant improvement in early lifecycle defect detection and removal to eliminate some, or all, of the unnecessary rework.

Before discussing software quality any further, it is important to clarify a few important terms: error, fault, and failure. Unfortunately, the software engineering literature often contains contradictory definitions of these terms. In fact, IEEE standard 610.12-1990 provides four definitions of the terms error, ranging from "incorrect program condition" (referred to as a program error) to "mistake in the human thought process" (referred to as a human error) [9]. To allay confusion, we provide a definition for each term that will be used consistently throughout this dissertation. These definitions were originally given by Lanubile, et al. [10], and are consistent with software engineering textbooks [11, 12, 13] and IEEE Standard 610.12-1990 [9]:

- Error – defect in the human thought process made while trying to understand given information, solve problems, or to use methods and tools. In the context of software requirements specifications, an error is a basic misconception of the actual needs of a user or customer.
- Fault – concrete manifestation of an error within the software. One error may cause several faults, and various errors may cause identical faults.
- Failure – departure of the operational software system behavior from user expected requirements. A particular failure may be caused by several faults and some faults may never cause a failure.

The term defect is a generic term used to describe any of these three types of problems. The definition of an error used in this dissertation more closely relates to

the human error definition rather than the program error definition in IEEE Standard 610.12-1990.

The main drawback of software inspection techniques that focuses exclusively on faults is that the underlying cause of the fault (i.e., the error) is neither addressed nor identified. Error taxonomy can help developers detect and eliminate errors and related faults. Furthermore, by identifying errors, developers can find additional related faults that may have been overlooked (similar to a doctor finding and treating all symptoms once he/she knows the underlying disease). Therefore, an error-based inspection process is needed.

The idea of using error information to improve software quality is not novel. Researchers have used information about source of faults in different ways. Some techniques that focus on errors determine the cause of only a sample of previous faults to suggest software process changes and prevent future defects [14, 15, 16, 17, 18, 19, 20]. In the cases where techniques do address the underlying cause of faults (e.g., Root-Cause Analysis [17], Orthogonal Defect Classification [21], and faults Error Abstraction [10]), the research has focused primarily on errors from the software engineering domain. These approaches lack a strong cognitive theory to describe the types of mistakes made when creating software artifacts. Human Error research in cognitive psychology builds upon theoretical models of human reasoning, planning, and problem solving, and how these ordinary psychological processes fail [22, 23, 24, 25, 26]. The exploitation of human error research broadens our understanding of errors that software engineers make during development.

To address this issue, Walia and Carver have combined the information from software engineering and cognitive psychology to develop requirements error taxonomy [27]. Walia et al., have also evaluated the usefulness and completeness of the taxonomy with a family of four controlled empirical studies [27, 28, 29, 30].

The results from the empirical studies conducted by Walia and Carver [27, 28, 29, 30] at *Mississippi State University* and *North Dakota State University* show that the requirement error taxonomy improves the defect detection effectiveness of both individual inspectors and teams significantly as compared to the fault checklist-based inspection process. A second important value of the requirement error taxonomy is that it can focus developer's attention on common errors during the requirement engineering process. An awareness of these common errors makes developers less likely to commit them and more likely to create an artifact that will have fewer defects to remove during the review and testing. Walia and Carver [31] have investigated the usefulness of the requirement error taxonomy as a defect prevention technique. A controlled study with university students showed that the developers can avoid making errors if they have a *priori* information about the types of errors that can occur during requirement development. Section 2 provides a brief description of the error taxonomy along with its development and evaluation processes.

This Master's paper focuses on developing an automated tool which is intended to incorporate the error abstraction process and the requirement error taxonomy while replacing the existing labor intensive manual "*error inspection process*" of identifying defects in requirements with intent to improve the inspector's efficiency and effectiveness.

The process of identifying defects using the automated tool will be characterized by: individual preparation done by authors, confined roles of inspectors, preparing themselves by using tutorials, step-by-step guide on identifying faults, classifying and abstracting errors from faults in time by understanding requirement error taxonomy, which are then tabulated as Fault list, Error list, and New Fault list. It is highly likely that some faults or errors could go unnoticed during 1^{st} cycle of inspection which could be further brought under lens by repeating the inspection cycles (2^{nd}, 3^{rd}, 4^{th} ...).

The automated software inspection tool is designed to ensure that the inspectors follow the defect detection process of locating faults, abstracting errors from faults, and re-inspecting for faults overlooked during the first inspection, and making sure that the software review process is being visible to the moderators and the managers that will help them make decisions regarding the software quality measurement.

The later sections of this paper, compares different existing software inspection tools available in the market. This is followed by the discussion of an automated software inspection tool that was developed to support the error inspection process.

2. RELATED WORK

This chapter is devoted to covering the knowledge in order to better understand the software faults, error abstraction process and the requirement error taxonomy, and a survey of existing software inspection tools.

2.1. Software Inspections

Software inspection process was first introduced by Michael E. Fagan in 1970 which resulted as part of software development while he was employed at IBM. In order to differentiate software inspection from general inspection software inspection should be called the in-process inspection [1]. Inspection is a static analysis method used to verify quality properties of software products. In other words they are a means by which it enables verifying intellectual products by manually examining the product during its software life cycle for finding and eliminating defects [2]. Software inspection when applied in the early stages of software life cycle namely requirements, design, coding proves more beneficial as defects that propagate from one stage to other can be avoided. If we neglect to perform inspections and if defects are missed in requirements phase, it would get amplified in design phase and likewise it will get even more amplified in the coding stage. The earlier the defect is found, the lower is the cost, and the easier is to fix. This also ensures that we have the correct base for further stages of software life cycle enabling developers to produce a high quality software product with good quality. It is beneficial if a small group of peers dedicate themselves in finding defects in one stage at a time while maximizing the defects found rather than each individual concentrating to find defects each one of different stages. Experiences with software inspections have shown that time spent

to accommodate inspection process in the software life cycle has helped in gaining time during the testing and manufacturing phase and saved rework efforts.

2. 2. Software Faults

In a generic sense, faults arise when the development work being done does not match the software specification already developed or would cause problems downstream as shown in Figure 2.1.

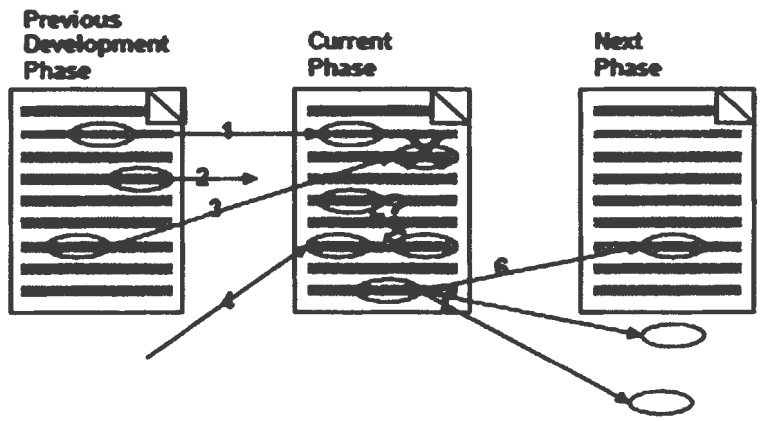

Figure 2.1. Fault transformation from phase to phase.

1. ***Information transformed correctly***: Figure 2.1 shows information is transformed correctly from previous development phase to current phase, which is represented by arrow 1
2. ***Information lost during transformation***: Figure 2.1 shows that some information is lost during the transition phase of the project from development to current phase. In figure it is represented with arrow 2 going halfway between the previous development phase and current phase.
3. ***Information transformed incorrectly***: Incorrect information is passed from previous to current phase, if that incorrect information is used it will snowball

problems in future. It is shown in figure 2.1 with a crossed use case with arrow 3 followed from previous phase.

4. ***Extraneous information introduced***: Introducing extraneous information which is not in scope or software specification can also cause problems downstream. It is represented with arrow 4 pointing to current phase.
5. ***Multiple inconsistent transformations occurred for same info***: Multiple inconsistent transformations can lead to confusion and difficulty in understanding data. As a result it is highly likely to have defects in such instances. It is represented with arrow 5 in the current phase with '?' symbol.
6. ***Multiple inconsistent transformations possible for same info***: It is possible that the same information can be transformed inconsistently between two phases as shown between current phase and next phase. In figure 2.1 it is represented with arrow 6 having transformations between current phase and the next phase with only one information passed on to next phase.

2.3. Background on Error Abstraction and Requirement Error Taxonomy

Nine different methods have used causal analysis to determine the source of a fault and suggest preventive actions (e.g., [14, 19]) or process changes (e.g.,[15, 18, 20, 32]). These methods were successful relative to their goals, but were incomplete because they focused on a representative sample of faults (potentially overlooking many errors). Nevertheless, the insights provided by these methods provided input to the requirement error taxonomy. A complete discussion of these methods, their limitations and their

contributions to the requirement error taxonomy has been published in a systematic literature review [27].

Lanubile et al., proposed the Error Abstraction approach, in which developers analyze faults detected during an inspection to determine the underlying errors likely to have caused them. These errors are then used to guide a re-inspection to detect additional faults. This work produced some promising initial results, but Lanubile, et al., did not pursue this research [10]. Walia and Carver work build their work on Lanubile's approach by formalizing requirement error taxonomy, with additional input from cognitive psychology, to better support developers during the error abstraction and re-inspection process. Figure 2.2 illustrates their previous research in developing and evaluating the requirement error taxonomy. This work is briefly discussed in Section 2.1.

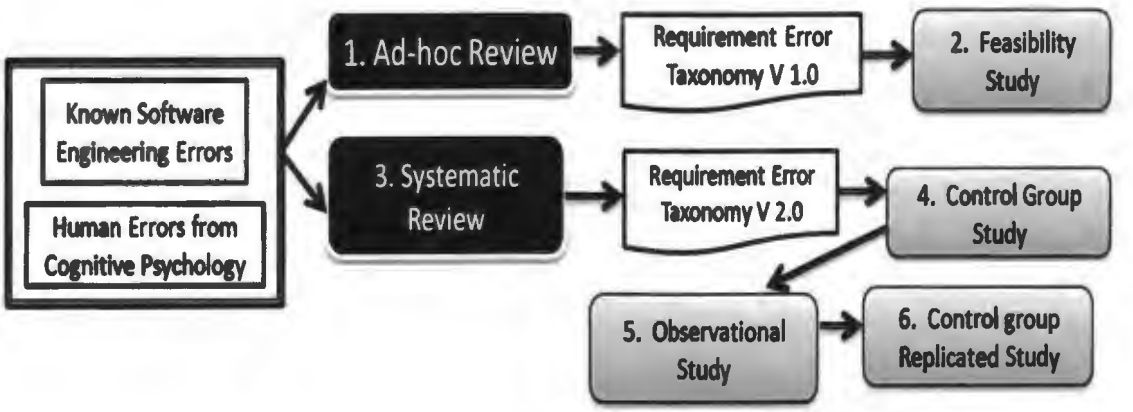

Figure 2.2. Process of developing and evaluating requirement error taxonomy.

2.3.1. Development of Requirement Error Taxonomy

The requirement error taxonomy has evolved through two versions. To create the initial version (V1.0), Walia and Carver performed an ad-hoc review of the software engineering and psychology literature to identify and classify requirement errors [33].

Next, this taxonomy was empirically evaluated to determine its usefulness to support the error abstraction and re-inspection process [28]. After establishing the feasibility of such an approach, a more formalized, systematic literature search of software engineering and cognitive psychology research was performed to refine the error taxonomy. The systematic review, commonly used in medicine, is a process for documenting high-level conclusions that can be derived from a series of detailed studies [34]. The systematic review identified 149 papers (from software engineering, human cognition, and psychology) that provided insights into evolving the requirement error taxonomy into V2.0 [27].

The errors identified from software engineering and cognitive psychology research were analyzed for similarities, and grouped into fourteen error classes (as shown in Table 2.1). These error classes were then classified into three high-level error types: People Errors (arise from the fallibilities of the people involved in the development process), Process Errors (arise when selecting the appropriate processes for achieving the desired goals, relate mostly to the inadequacy of the requirement engineering process), and Documentation Errors (arise from mistakes in organizing and specifying the requirements, regardless of whether the developer properly understood the requirements).

To illustrate the information contained in the error taxonomy, an example participation error (one of the People Errors) along with related faults is described here:

- Error: An important stakeholder (e.g., a bank manager in an ATM system) was not involved in the requirement gathering process.
- Fault: Some functionality (e.g., handling multiple ATM cards simultaneously at different machines) was omitted.

The complete systematic review process, the organization of the requirement errors into the error taxonomy, and the details of the requirement errors (along with examples of errors and faults) can be found in the systematic review publication [27].

Table 2.1. Description of requirement error classes.

	Communication	Poor or missing communication among the various stakeholders
	Participation	Inadequate or missing participation of important stakeholders
	Domain Knowledge	Requirement authors lack knowledge or experience with problem domain
	Specific Application Knowledge	Requirement authors lack knowledge about specific aspects of the application
	Process Execution	Requirement authors make mistakes while executing requirement elicitation and development, regardless of the adequacy of the chosen process
	Other Cognition	Other errors resulting from the constraints on the cognitive abilities of the requirement authors
	Inadequate Method of Achieving Goals and Objectives	Selecting inadequate or incorrect methods, techniques, approaches to achieve a given goal or objective
	Management	Inadequate or poor management processes
	Elicitation	Inadequate requirements elicitation process
	Analysis	Inadequate requirements analysis process
	Traceability	Inadequate or incomplete requirements traceability
	Organization	Problems while organizing requirements during documentation
	No Usage of Standard	Problems resulting from the lack of using a documentation standard
	Specification	General documentation errors, regardless of whether requirement author correctly understood the requirements

2.3.2. Evaluating the Requirement Error Taxonomy for Detecting Defects During Inspections

Walia and Carver have evaluated the usefulness of the requirement error taxonomy with four empirical studies. The validation goal of these studies was to ensure that: 1) the

error classes are clearly described, useful, and complete, and 2) the developers can use the error taxonomy to increase their defect detection effectiveness during inspections.

Study 1 and 3 (see Figure 2.2), were conducted in senior-level capstone courses where students developed a project for real customers. In these studies, the students first performed an inspection of their requirement document to identify faults. Then, they were trained on the use of error taxonomy. The students then used the error taxonomy to abstract and classify the errors that caused the observed faults. Finally, the students used the error information to guide the re-inspection of the requirement document. The results from these two studies indicated that the participants found the error taxonomy both easy to use and effective. In addition, by using the error taxonomy, the participants found a significant number of new faults during the re-inspection. Finally, most participants found errors that were derived from the cognitive psychology human error research, 10%-20% of the total errors reported [28, 30, 33].

Study 2 and 4 (see Figure 2.2), were conducted with students enrolled in graduate level courses. In these studies, one group of students (i.e., the experiment group) used the same procedure as in the Study 1 and 3 described above. The other group of students (i.e., the control group) inspected the artifact two times without using error abstraction. In Study 2, the control group participants used the same fault inspection technique during both inspections and in Study 4, they used a more mature fault inspection technique for the re-inspection. The results from the experimental group were compared with the results from the control group to determine what portion of the additional faults found during the re-inspection can be attributed to the use of the error abstraction and classification approach.

The results from these studies showed that the group who used the error abstraction and classification process found significantly more faults during re-inspection than the control group, providing more evidence of its usefulness [29, 30].

The results from these four studies can be summarized as follows: 1) the error abstraction and classification approach improves the effectiveness (number of faults found) of inspectors during a requirements inspection, 2) the requirement error taxonomy is subjectively useful for inspectors to find errors and faults, and 3) the human error research from cognitive psychology helped inspectors detect more faults. More details of the experiment designs and results from each of these studies can be referred.

While the requirement error taxonomy has been effective in detecting defects during inspections, a more useful analysis required evaluating the effectiveness of the requirement error taxonomy for preventing defects from occurring during the requirements development. Leape, and other researchers have employed a similar approach to the analysis of adverse medical events in order to understand what caused the individuals to make errors [22, 23]. Leape et al., argued that the underlying cause of the problems should be used to prevent errors rather than attempting to remove the errors. Because the errors are mistakes or misunderstandings of the software engineers while creating a software artifact, the information about the commonly made errors can be used to educate software engineers to prevent them from making errors in the first place.

Defect prevention techniques can be used during the creation of software artifacts to help developers create high-quality artifacts. These artifacts should have fewer faults that must be removed during inspection and testing. Requirement Error Taxonomy also helps focus developers' attention on common errors that can occur during requirements

engineering. Walia and Carver claim that, by focusing on those errors, the developers will be less likely to commit them. They have investigated the usefulness of the Requirement Error Taxonomy as a defect prevention technique. The goal was to determine if making requirements engineers' familiar with the Requirement Error. Taxonomy would reduce the likelihood that they commit errors while developing a requirements document. They conducted an empirical study in which the participants were given the opportunity to learn how to use the Requirement Error Taxonomy by employing it during the inspection of a requirements document. Then, in teams of four, they developed their own requirements document. This requirements document was then evaluated by other students to identify any errors made. The hypothesis was that participants who find more errors during the inspection of a requirements document would make fewer errors when creating their own requirements document. The overall result from their experiment supported this hypothesis and provided the motivation to further investigate the promise of using the error taxonomy as a defect prevention technique [31].

2.4. Existing Software Inspection Tools

There have been many tools developed to help inspectors during the software inspection process. All of these tools have been tailored better to support a desired software inspection process in one way or the other. For example, an inspection tool supports inspection of documents written in natural language, another tool supports inspection of source codes in C and C++ programming languages, and another tool was developed to support the distributed inspections by having a web centric program which enables different group of inspectors (who are scattered across the globe) to inspect a particular

software artifact. Some of the existing inspection tools and techniques are discussed below along with their relationship to this master's paper work:

2.4.1. Asynchronous or Synchronous Software Inspection Tool (ASSIST)

ASSIST is an Asynchronous or Synchronous software inspection tool designed by F. Macdonald which was developed based on client/server architecture [35]. ASSIST was designed for inspecting any kind of documents. This tool supported individual as well as group-based phased inspections. One good thing about group-based inspections was inspectors had a choice of either performing inspections in the same place or different place using synchronous meetings. Group-based inspections can be either synchronous or asynchronous in nature. Perhaps keeping in view of all the inspection processes, Macdonald came up with a common software inspection template which could withstand future advances in the inspection processes. Thus this inspection template was converted into a process definition language known as IPDL (inspection process definition language) and further embedded into ASSIST. These are some of the features of ASSIST [35].

- It aids to find defects.
- Features for enabling metric collection and analysis of collected values.
- Enables distributed inspection process.
- An online checklist to check each item in the list as and when they are finished.
- There is a provision of text browser which can be used to comment on the documents and make annotations, which will explain what defects have been found in the document during the inspection process.

2.4.2. Scrutiny

Scrutiny is a web-based software inspection tool which is designed to support distributed inspections. It was developed in by Bull HN Information systems together with University of Illinois [35]. The process of inspection using scrutiny is divided in four different phases namely Initiation, Preparation, Resolution, and Completion.

In the initial phase formation of inspection team occurs followed by preparation of necessary documentation for inspection process by Moderator. In the preparation phase the inspectors go through the documents presented to them by moderators and annotate them. In the resolution phase all the inspecting stakeholders meet and get an insight of the inspection results and their findings via inspection. The final Completion phase involves reworking as well as following up of phases in the inspection process. Scrutiny tool is designed to support only text documents which can be feature enhanced for further development. It does not support checklists and comprehension [36].

2.4.3. ICICLE

ICICLE stands for Intelligent Code Inspection in a C Language Environment. ICICLE, as the abbreviation suggests, is designed to assist C language code inspection and support a set of complex tasks performed during code inspection process [37]. This tool helps inspectors in finding the common defects by itself with the help of a rule-based static debugging tool and the UNIX lint tool. It was designed to replace the manual process of inspecting code which traditionally is done manually using and pen and paper to list all the errors found in the code. This process was very tedious, error prone, and inconsistent. ICICLE takes care of all these problems.

ICICLE, which concentrates on eliminating aforementioned difficulties, operates with the following attributes [37]:

- It has an intelligent mechanism of inbuilt tool, which can find errors which are commonly made and thus makes it much easier and reduces burden for inspectors of finding errors. Instead they can resort on verifying correct implementation of requirements, specifications and designs.
- It caters different knowledge base for inspecting code such as domain knowledge, environment knowledge, and a source of analysis namely cross-referencing.
- It lets inspectors to surf over source code, in a windowed environment which rather saves time from having them to go through a hard copy of several files.
- A shared window which enables inspectors to share comments for findings and discuss them in meeting rather than having papers distributed for following the reader.
- The process of inspection using ICICLE is divided into two phases. One being the individual inspection and the other inspection meeting. A feature for writing comments for every line of code during the inspection process is enabled for inspectors. A responsive referencing system is designed for supplying variables and functions with a quick movement across the lines of code.

2.4.4. CSI

Collaborative Software Inspection is a web-based inspection tool which support distributed inspections. It makes it easier for distributed inspection process as all the documents and materials required for inspection are hosted online [35]. It was designed in order to support four kinds of collaborative inspection meetings namely:

a) Same time, same place

b) Same time, different place

c) Different time, same place

d) Different time, different place

In this process every inspector participating in the individual inspection finds and creates a list of faults and hands it over to the author who initially created documents. And it is author who draws a parallel to all faults found by inspectors and to address them in a group meeting.

CSI provides an online web browser that can effectively give details about data being used in inspection and status of it. It auto numbers each line in the document; also it lets inspectors to write comments for that particular line by an annotation window. There is a provision of hyperlinks from inspected material to a fault list. One may feel that it is insufficient to have annotations since it is confined to only specific lines. But there is a notepad available on general inspection documents.

2.4.5. InspeQ

Inspecting Software in phases to ensure Quality is shortly named as InspeQ. Knight and Meyers developed this inspecting toolset to support their phased inspection technique [38, 39]. Knight and Meyers together developed this inspection technique which promotes an inspection process to be "rigorous, tailor-able, efficient in its use of resources, and heavily computer supported called phased inspections" [38]. Phased inspections are so designed that after completion of each and every phase of inspection it is expected that the software product has a minimum set of properties for which it was inspected. One cannot declare phased inspection is complete without showing that product satisfies all the checks.

Inspectors, given the task to perform phased inspections are well explained about what their role is and objectives are. As a result it is made easier for them to walk on the lines of predefined phases which are supported by computers. There are two phases in InspeQ: single-inspector, multiple-inspector. Single inspector as the name suggests is performed by single inspectors who perform rigorous checklist inspection. It's a phase in which the product can comply to one or all of the checks listed in the checklists. One cannot move to the next phase if even one of checks is not satisfied.

Not always single-inspector phase proves handy as there are times when it cannot determine properties of products for next phase in such instances use of multi-inspector phase helps. In this phase individual checking is done by individual inspectors against a set of checklists and once its completed they meet to discuss their findings which could give a lead for fault findings. InspeQ is a great tool which aids inspectors in conducting phased inspections effectively and efficiently. The computer support provided by the tool helps phased inspections to be performed quickly along with mechanisms which could keep a check on the predefined process if being followed or not and there are features in the tool which lets one evaluate the results of inspection.

2.4.6. WiP

It is common to have teams across the globe for a company which has a global presence. With the advent of web-based technologies lives had become easier for many companies which encouraged working collaboratively effectively and efficiently in a distributed eco-system. WiP was one such tool which incorporated the entire stand-alone work environment into a web-based environment with the help of world-wide-web. It provides online features such as document handling for inspectors; it would let inspectors

annotate online documents by not manipulating the document itself as all the annotations were stored in a server to avoid multiplication of data. It would allow taking simple inspection statistics too. The initial development motive of WiP was to find if really an inspection process can be carried out via World Wide Web [36].

2.4.7. Review Pro

Review Pro, a web-based, software technical review, inspection tool was developed by Software Development Technologies Corporation. This was developed keeping in mind the importance of defect detection in the early stages. As a result the whole process was automated to effectively find defects and save time. Though web based this was a tool which was quite independent of web browser, web, messaging server software, and could be run in Windows NT and UNIX server platforms [35].

2.4.8. CheckMate

This tool works against a predetermined coding policy to inspect coding in C and C++ programming languages concentrating on classes and methods of the program. The coding policies could be custom set as needed by the inspectors. It has features to assess software metrics.

2.4.9. Limitations

Perhaps, one of the notable deficiencies in all the tools so developed is that it supports only textual documents. Even though text is the main type of documents used for inspection it should also support other type of documents. There can be diagrams for inspection too, so files with diagrams should also be allowed to be inspected. During the inspection process all the defects found are logged in papers. Therefore, there is a high

chance of misplacing or losing them. Inspectors are not given prior training to find defects and classify them. They are usually web centric tools.

It should be noted that all the tools developed so far are based on a predetermined framework for conducting inspection processes some identify defects in C, C++ programming language code, some find faults in documents which could be annotated, and some others are developed to do software inspection in web based environment. It becomes clear that the majority of tools are custom made to suit specific purposes for methods of inspection process. Likewise, the tool in this paper is designed to replace a predetermined labor intensive manual way of conducting error inspection processes. Considering some of the limitations of the existing tools, we have developed an automated software tool for error inspection process. The phases of finding faults, abstracting errors from faults, and classification of errors from the requirement checklist, which forms the basis of the tool are explained in the following section along with types of software defects and some examples of defects in requirements.

3. RESEARCH TOOL

3.1. Introduction

Inspection is an effective verification and defect detection process. The main goal of inspection is to find and fix defects and not defect prevention. The table 3.1 below describes different types of software defects that can be found during an inspection:

Table 3.1. Software defect types.

Type	Description
Omission	Necessary information about the system has been omitted from the software artifact.
Incorrect fact	Some information in the software artifact contradicts information in the requirements document or the general domain knowledge.
Inconsistency	Information within one part of the software artifact is inconsistent with other information in the software artifact.
Ambiguous Information	Information within the software artifact is ambiguous, i.e. any of a number of interpretations may be derived that should not be the prerogative of the developer doing the implementation.
Extraneous	Information is provided that is not needed or used.
Miscellaneous	Other defects; e.g. a requirement may be found in an inappropriate section of the document.

To better understand the process of detecting faults in requirements, this section describes examples of different type of software faults using specifications for a Gas Station Control System (GSCS). Examples (3.1, 3.2, 3.3) has excerpts from requirement specifications with their defects addressed. The key point which leads to defect is underlined and 'Note' explains about defect found.

Example Requirement: Gas Station Control System (GSCS)

Overview:

- "... The gas station allows customers to purchase gas (self-service) or to pay for maintenance work done on their cars. Local gas stations may have billing accounts set up so that the gas station is sent a monthly bill, rather than paying for each transaction at the time of purchase. There will always be a cashier on-duty at the gas station to accept cash payments or perform system maintenance, as necessary. Customers have the freedom to use visa/master cards."

- The requirements in this excerpt *"...concern how the system receives payment from the customer. A local customer has the option to be billed automatically at the time of purchase, or to be sent a monthly bill and pay at that time. Customers can always pay via cash or credit card (visa/master). "*

Example 3.1: Functional Requirement 5

If payment is to be made by cash, the cashier is responsible for accepting the customer's payment and making change, if necessary. When payment is complete, the cashier indicates this on the cashier's interface. The GSCS and the gas pump interface then return to the initial state.

☞ *Note: Information was lost during the creation of the requirements. As the description does not mention clearly, what is the purchase price? To handle a cash transaction, the cashier must know what the purchase price was and how greater a cash payment can be accepted. This information has been left out of the description of the functionality - therefore we have a defect!*

Example 3.2: Functional Requirement 3

If the customer has selected to pay at the time of purchase, he or she can choose to pay by cash or credit card. If the customer selects cash, the gas pump interface instructs the customer to see the cashier to pay at cash counter If the customer selects credit card, the gas pump interface instructs the customer to swipe his or her credit card through the credit card reader. If an invalid or no selection is made, the GSCS will use the credit card payment option, which is the default.

☞ *Note: Information was translated incorrectly. In the example, domain knowledge should indicate that defaulting to credit card payment is an incorrect response. (What kind of transaction ever happens this way?) Because we know that this functionality should not be implemented the way it is described, we have a defect.*

Example 3.3: Functional Requirement 2:

After the purchase of gasoline, the gas pump reports the dollar amount of the purchase to the GSCS. The maximum value of a purchase is $999.99. The GSCS then causes the gas pump interface to query the customer as to payment type.

Functional Requirement 4:

If payment is to be made by credit card, then the card reader sends the account number to the GSCS. If the GSCS receives an invalid card number, than a message is sent

to the gas pump interface asking the customer to swipe the card through the card reader again and if still doesn't accept it look for cashier. After the account number is obtained first enable pump, pump gas or hang-up hose to disable pump and get purchase price, the account number and purchase price are sent to the credit card system, and the GSCS and gas pump interface are reset to their initial state. The purchase price sent can be up to $10000.

☞ *Note: Information was described inconsistently. Because we don't know from domain knowledge which of the two descriptions is correct, we have found a defect.*

3.2. Defect Detection Cycle

Now that we have seen some examples of defects, it is important to understand the underlying steps adopted in inspecting requirements checklist and to know on what basis defects found are classified and recorded in corresponding tables of each step that stands as a foundation for the automated tool developed in this paper (figure 3.1).

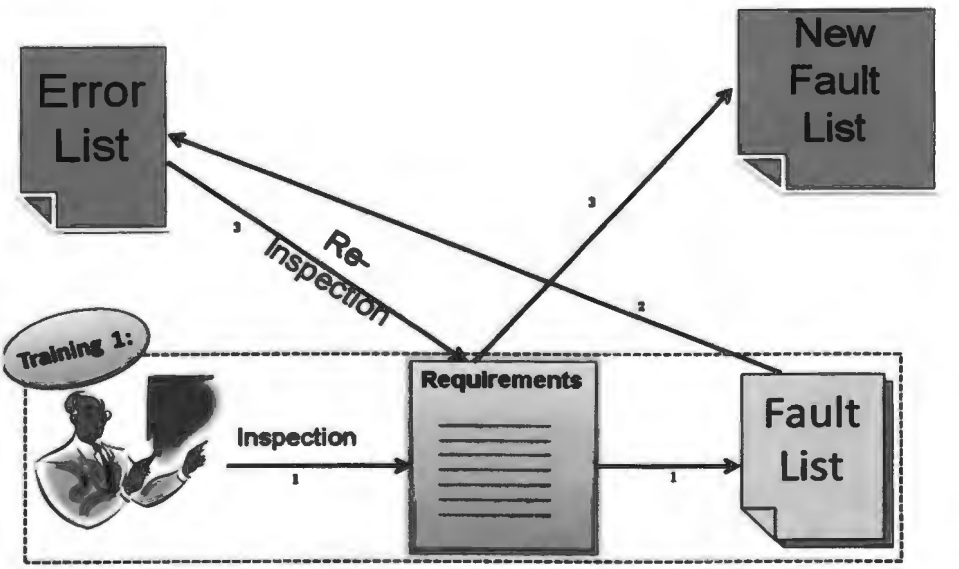

Figure 3.1. Diagrammatic representation of three steps in defect finding cycle.

Step 1: Finding Faults Using Fault Checklist.

Inspectors undergo training on how to find faults using the fault checklist technique. They read the requirements document and use the knowledge gained from the training to find faults and log them in fault list.

Step 2: Finding Error From Fault List

Inspectors are trained on the requirement error taxonomy and on how to abstract errors from faults using the error information in the requirement error taxonomy and fill error list form. Once they are trained they use the knowledge gained from training to extract errors from faults on their fault lists and log the extracted errors on error list form. A detailed description of the error abstraction training is published in [9].

Step 3: Find New-Fault List.

Now that inspectors have gathered errors from the above step they use the error information from Step 2 to re-inspect the requirements document to find more faults. And the additional faults found are logged in error- fault list (new fault list).

3.2.1. Fault List

How does one detect fault?

- By reading the document
- By understanding what the document describes
- By answering the questions in the fault checklist

Different checklist techniques with their characteristics which are used to detect defects to fill fault list form are given below:

3.2.1.1. General Faults (G)

- Are the goals of system defined?

- Are the requirements clear and unambiguous?
- Is a functional overview of system provided?
- Is an overview of operational modes provided?
- If assumptions that affect implementation have been made, are they stated?
- Have the requirements been stated in the terms of inputs, outputs, and processing for each function?
- Are all functions, devices, constraints traced to requirements and vice versa?
- Are the required attributes, assumptions and constraints of the system completely listed?

3.2.1.2. Omission Faults

- *Missing Functionality (MF)*
    - Are the desired functions sufficient to meet the system objectives?
    - Are all inputs to a function sufficient to perform the required function?
    - Are undesired events considered and their required responses specified?
    - Are the initial and special states considered (e.g., system initiation, abnormal termination)?
- *Missing Performance (MP)*
    - Can the system be tested, demonstrated, analyzed or inspected to show that it satisfies the requirements?
- *Missing Interface (MI)*
    - Are the inputs and outputs for all interfaces sufficient?
    - Are the interface requirements between hardware, software, personnel and procedures included?

- *Missing Environment (ME)*
    - Have the functionality of hardware or software interacting with the system been properly specified?

3.2.1.3. Commission Faults

- *Ambiguous information (AI)*
    - Are the individual requirements stated so that they are discrete, unambiguous, and testable?
    - Are all mode transitions specified deterministically?
- *Inconsistent information (II)*
    - Are the requirements mutually consistent?
    - Are the functional requirements consistent with the overview?
    - Are the functional requirements consistent with the actual operating system?
- *Inconsistent and Extra Functionality (EF)*
    - Are all desired functions necessary to meet the system objectives?
    - Are all inputs to a function necessary to perform the required function?
    - Are the inputs and outputs for all interfaces necessary?
    - Are all the outputs produced by a function used by another function or transferred across an external interface?
- *Wrong selection (WS)*
    - Are all the requirements, interfaces, constraints, etc. listed in the appropriate sections?

3.2.1.4. Other Faults (O)

- If you find additional faults, not related to specific questions on the checklist, which do not fall in any of the existing categories, classify it as Other (O).
- Once faults are found, it is logged in fault list form as shown in table 3.2 which has various fields.

Table 3.2. Fault list form.

Fault#	Page#	Req#	Fault Class	Description	Time Found	Importance Level	Probability of Causing Failure	Break

The fields described in Table 3.2 are listed follows:

- Fault #- serial identification number (e.g., 1, 2, 3, etc).
- Page #- maps to the page number in a SRS document where that fault is present (e.g., 3, 5, 6 etc).
- Requirement #- maps to a particular requirement number where a fault is found (e.g., FR2.1, FR3, etc).
- Fault class- describes the classification of a fault. A fault is classified in following classes using fault checklist: General (G), Missing Functionality (MF), Missing Performance (MP), Missing Interface (MI), Missing Environment (ME), Ambiguous Information (AI), Inconsistent Information (II), Incorrect or Extra Functionality (EF), Wrong Section (WS), Other (O).

- Description- provides a brief but clear description of the fault in the requirements document.
- Time found- it is the time when a particular fault was found.
- Importance level- this is the scale of importance of a particular requirement fault found during inspection and has to be classified as per following scale:
    - *0: not important, designer should easily see the problem*
    - *1: problem, if a failure occurs it should be easy to find and fix (e.g. change to 1 module)*
    - *2: important, if a failure occurs, it could be hard to find and fix (e.g. change to few modules)*
    - *3: very important, if a failure occurs, it could be very hard to find and fix (e.g., change to several modules and their dependencies)*
    - *4: if a failure occurs, it could cause a redesign*
- Probability of causing failure- describes the probability scale that a particular fault can cause system failure using following scale:
    - *0: will not cause fault of failure, regardless whether it is caught by the designer*
    - *1: will not cause fault or failure, because it will be caught by designer*
    - *2: could cause a failure, but will most likely be caught by designer*
    - *3: would cause a failure, will most likely not be caught by designer*
- Break: describes breaks taken during the inspection.

3.2.2. Error List

As mentioned in Section 1, as per IEEE standard terminology, Error is a defect in the human thought process made while trying to understand given information, solve problems, or to use methods and tools. In the context of software requirements specifications, an error is a basic misconception of the actual needs of a user or customer [9]. In order to fill up error list we have to abstract errors from fault list with the help of requirement error taxonomy and classify them. And the different requirement error classes were described earlier in table 2.1.

3.2.2.1. Error Abstraction

The error abstraction process helps to abstract errors/mistakes from the faults. Abstraction of errors can be done by the following steps:

- Analysis of the fault list
    - Why each fault (in your fault report form) represents a defect in the SRS?
- Grouping of the related faults
    - Group faults based on their categories or nature (e.g., G, MF, MP, MI, ME, AI, II, IF, WS)
- Eliciting the underlying reasons for the occurrence of the faults
    - Find pattern in the grouped faults and think of some believed reasoning for these faults to have occurred
    - Write down the errors (mapping errors to faults).

Also, inspectors use requirement error taxonomy that describes different types of errors that occur during development of the requirement.

Example 3.4, explains how error can be abstracted from requirements faults F1, F9.

Example 3.4:

Consider these faults:

Requirement Fault F1: The requirements say "The system keeps a rental transaction record for each customer giving out information and currently rented tapes for each customer." However, an explanation of exactly what information is given out for each customer has been omitted.

Requirement Fault F9: The requirements say that when a tape is rented, the "rental transaction file is updated." However, what it means to update the rental transaction file is not specified. The information to be stored here is not discussed.

Understanding of Error abstraction

RF1 and RF9 - can be classified as Missing Information (MI) class. The missing information about "How the information in the database is to be updated?"

Error can be that, "how the rentals are to be logged is not completely understood"

☞ *Note: however, it is not always the case that you will find an error responsible for multiple faults (as in above example). Error can be responsible for single faults, and patterns can also be found between errors in different classes*

Since, abstracting errors from faults is a very creative process, to support the error abstraction process; inspectors are trained on how to use the *Requirement Error Taxonomy* (that describes the different types of errors that can occur during the development of requirement document) to abstract errors from the faults during the error abstraction process.

Once errors are abstracted, use "Error list Form" as shown in table 3.3 to log errors corresponding to each fault (from your fault list that you submitted).

Table 3.3. Error list form.

Error #	Fault #	Description of Error	Time found	Break (time)

3.2.3. New-fault List

This step involves using the error information from the "Error List" to re inspect the SRS document for the faults that were missed during the first inspection. The process of re-inspection of software artifact using the error information includes following steps:

- For each error in the "Error List", inspect the SRS for fault(s) caused by it.
- For each new fault found, complete a row in the "New Fault List".
- An error can cause one or more faults.

Now that all three lists are explained it should be noted that lists should be filled in a sequential order starting from Fault List then Error List followed by New Fault List. A "New Fault List" form used by inspectors to log faults is shown in table 3.4.

Table 3.4. New fault list form.

Error#	Page#	Fault Class	Description	Time Found	Importance Level	Probability Of Causing Failure	Break

4. APPLICATION OF RESEARCH TOOL

The whole process of conducting software inspection taking different stakeholders into account like authors, and inspectors makes it feasible to design a software tool to cater error inspection needs. By understanding steps adopted for inspection as discussed in the earlier section this chapter presents an automated tool with some screenshots and description.

4.1. Assigning User Access

An author who conducts inspection process will be able to assign user name, password for inspectors, authors. Authors can assign temporary passwords and inspectors can change it once they login. Figures 4.1 to 4.3 show screens of adding users, changing passwords.

Figure 4.1. Adding new author window.

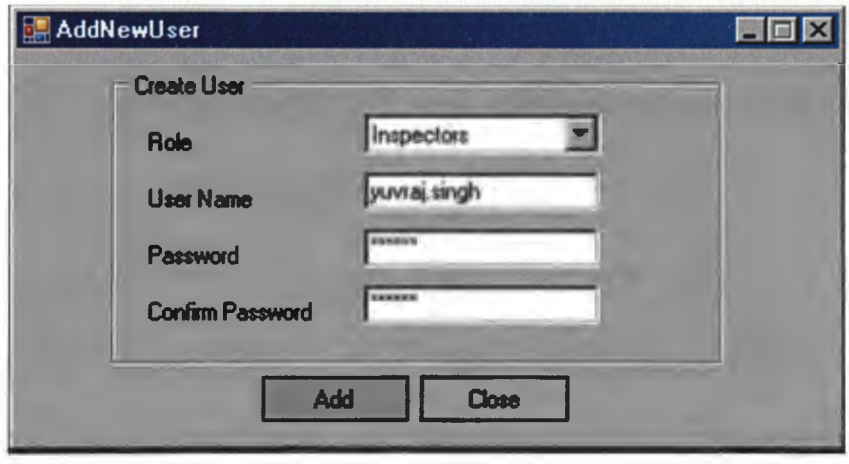

Figure 4.2. Adding new inspector window.

Once users login into their respective accounts they can change their temporary password to permanent.

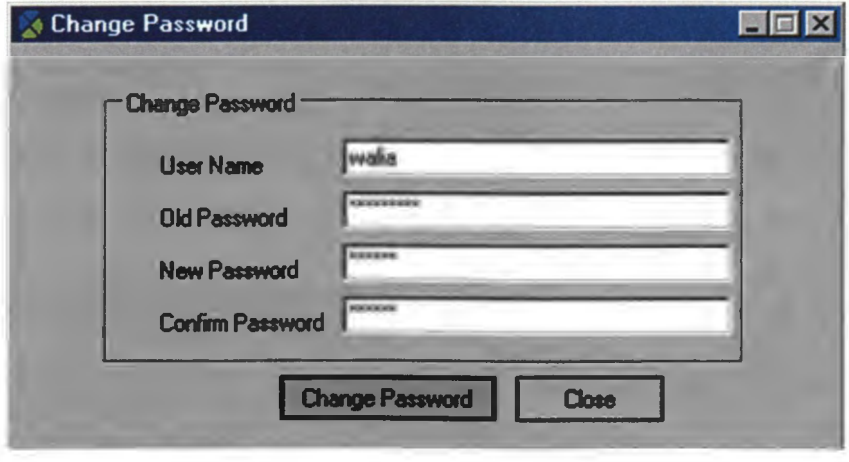

Figure 4.3. Change password window for changing user password.

There are different access rights assigned to author and inspector. Authors can host files and delete them but inspectors can only open the files hosted by authors but not delete them. Authors can view comments of inspectors recorded during inspection but they cannot

delete their comments. Authors do not need training for inspection but inspectors do need training for inspection. These differences in their profile is compared in Table 4.1

Table 4.1. Differences in author's profile versus inspector's profile.

Authors profile	Inspectors profile
Can assign authors or inspectors.	Cannot assign authors or inspectors.
Can upload and delete files for inspectors.	Can only view uploaded files by authors.
Can view inspectors name for whom file is uploaded.	Can authors name who uploaded the file.
Can view recorded comments from inspection.	Can record faults, errors, new faults from inspection.
Does not need training.	Inbuilt tutorial needs training for recording faults, errors, new faults.

4.2. Uploading a File for Inspectors

Authors can browse and upload requirement specification files for inspectors by selecting the names of inspectors in the Add New File window. To upload author needs to click on 'Browse' button then click open file, select inspectors to upload file by clicking on 'upload' button. AddNewFile window in figure 4.4 shows the list of inspectors who were added by author for inspection and can select one or more inspectors to upload file requirements checklists file (or any other file).

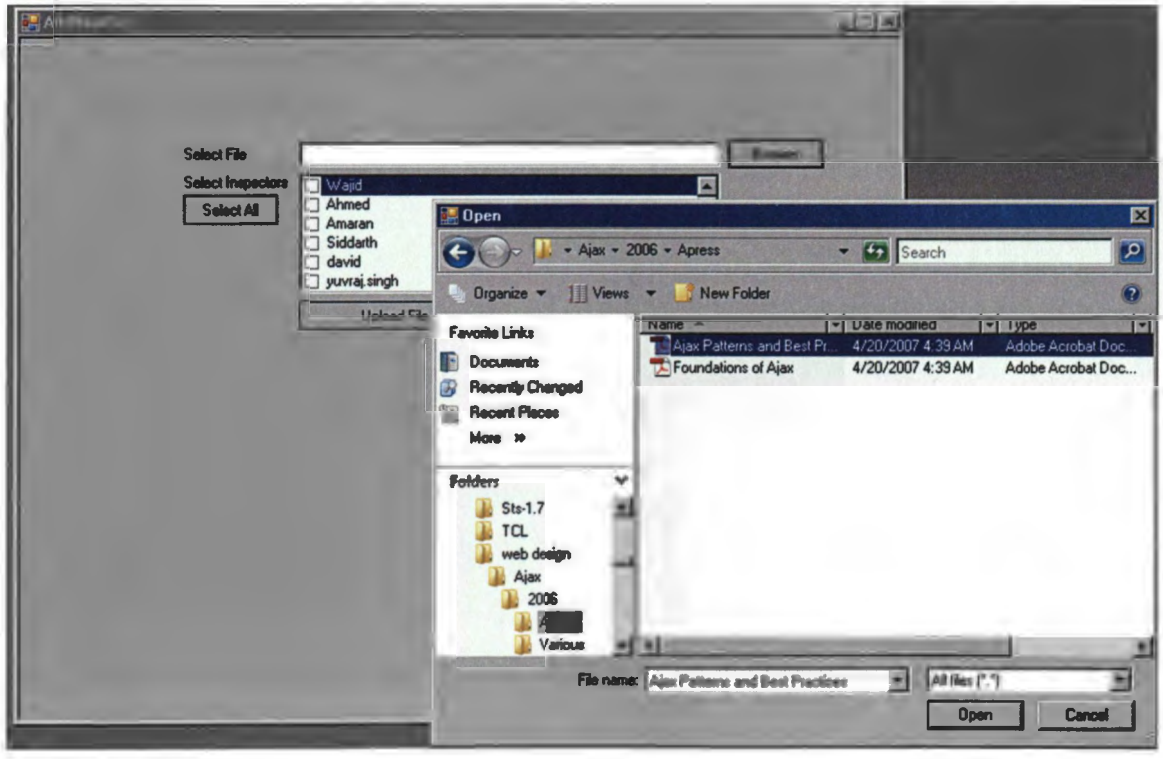

Figure 4.4. Add new file.

4.3. View Files and Forms

All documents hosted by authors for inspectors can be viewed. It can be checklist for them to validate and classify the defects in the inspection process. It shows the name of the author who uploaded it along with name of file with its extension (.pdf, .docx, .ppt, etc.,). Buttons for opening uploaded files are provided on the extreme right column as shown in figure 4.5. Like to 'view' button is available for recording faults and errors.

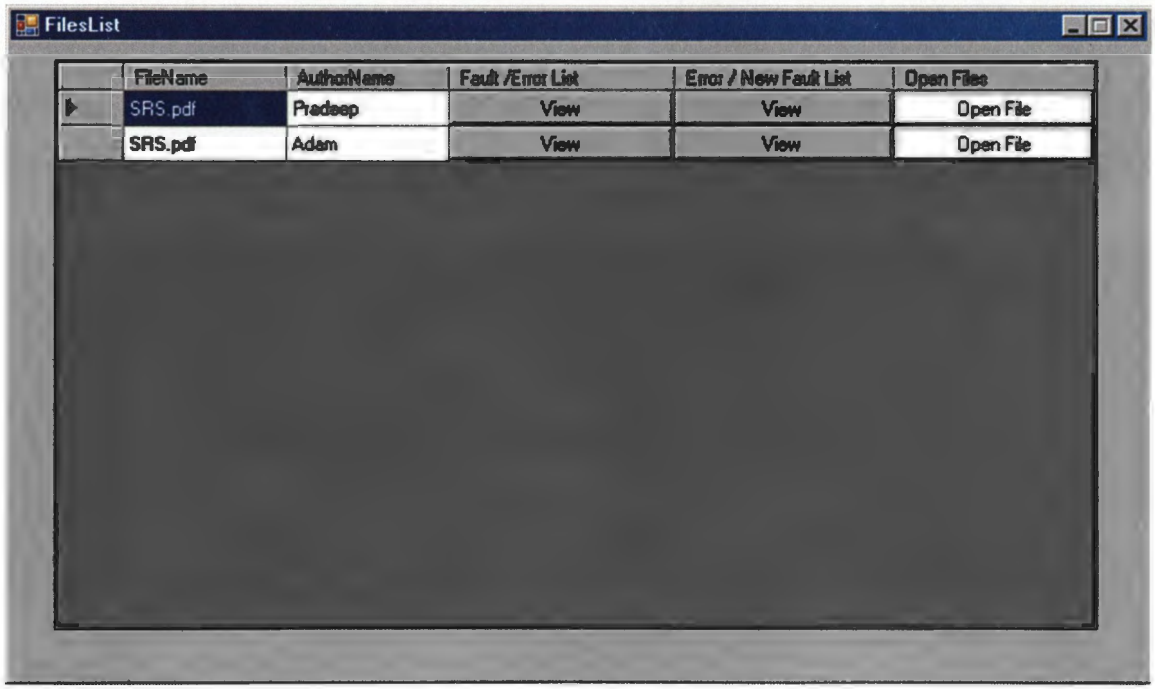

Figure 4.5. View files list.

4.4. Tutorials

Tutorials are provided for inspectors to self-learning and preparing themselves before proceeding to find defects. The figure 4.6 shows 'Tutorials' is listed under the 'Documents' tab.

Figure 4.6. Selection of tutorials.

Introduction to software inspections – This tutorial explains basic concepts and benefits of inspection, gives an insight of defect detection process and how is it practiced, a sample screenshot of tutorial is shown in figure 4.7.

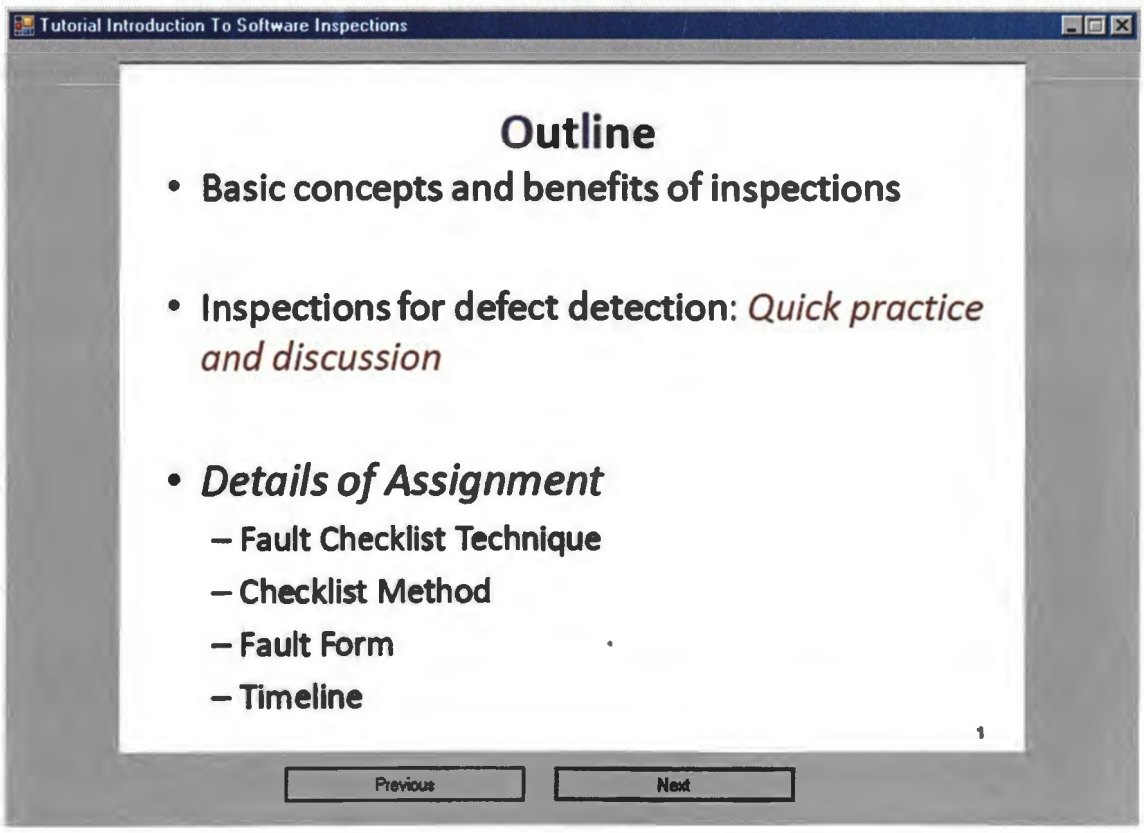

Figure 4.7. Sample screenshot of tutorial with previous and next button for changing slides.

Inspection Using Fault Check List Technique –This tutorial explains how to detect faults, different checklist methods, teaches how to classify the faults by using conditional checklist, and finally attributes of fault list form, what parameters should be used to fill the form with time constraints are explained with example.

Error Abstraction and Classification Process –. This tutorial explains how to abstract error from faults and classify them, while tabulating in Error list form. This also explains process

of finding new faults or more faults by re-inspecting software requirement specification (SRS) document using the errors already found.

Requirement Error Taxonomy – this document which explains different types of errors that occur during the development of requirement document supports the error abstraction process.

4.5. Fault/Error/NewFault List

Fault/Error List form is used by inspectors to log faults and errors during inspection as shown in figure 4.8. Inspectors can use '*Instructions*', 'Attributes' tab to fill the form.

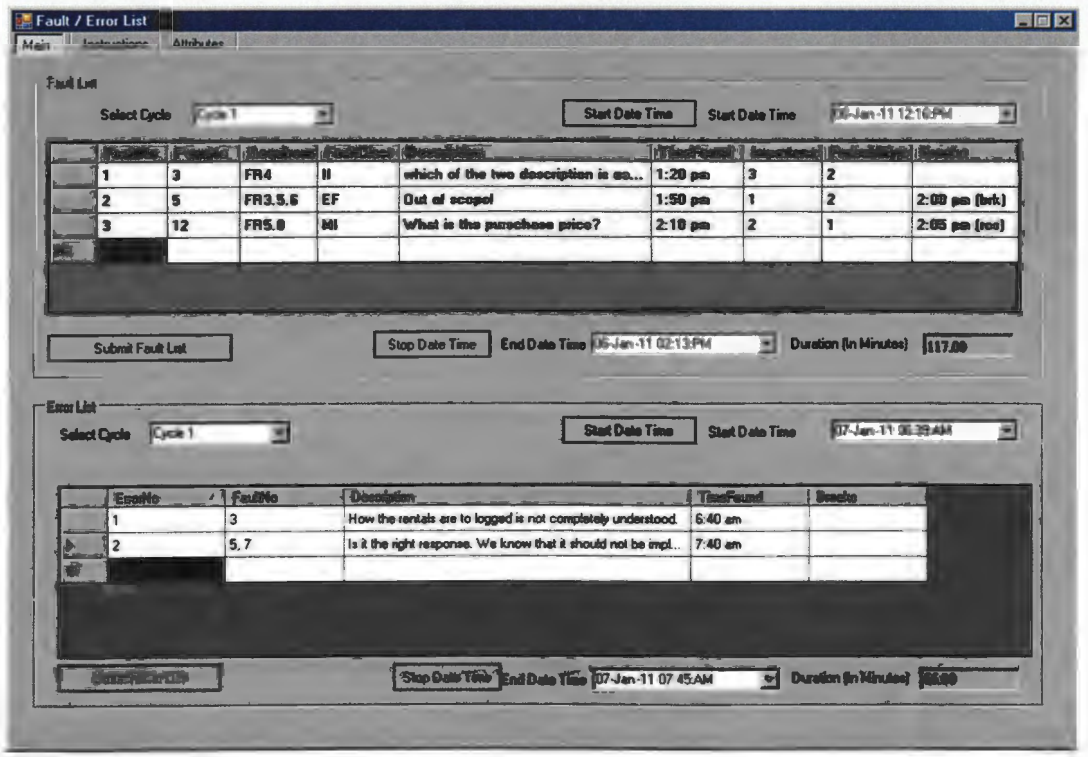

Figure 4.8. Shows fault list, error list.

The same 'Error list' saved from previous form is shown in the Error/New Fault List form for ease of use to find more faults and record in New Fault List as shown in figure 4.9.

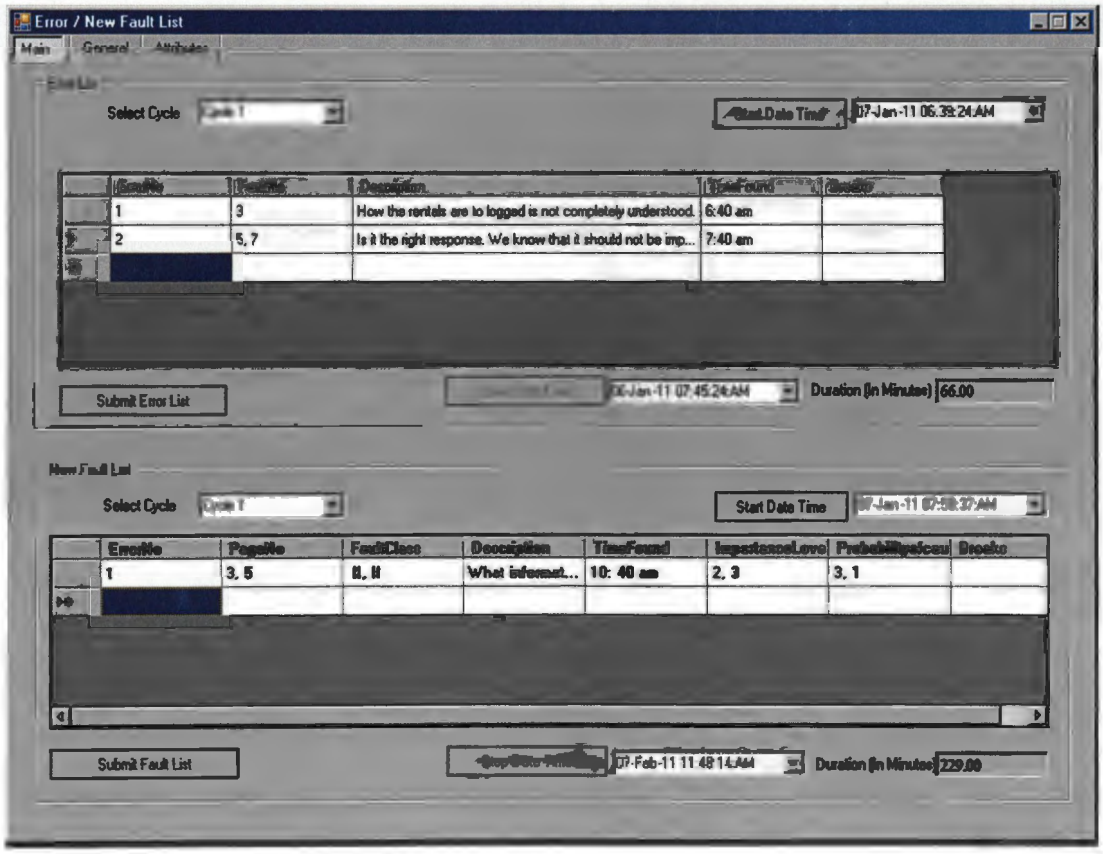

Figure 4.9. Shows error list and new fault list.

Fault list, Error list and New fault list are recorded in sequential order making it one inspection cycle. Likewise inspection can be repeated by keeping a count for each cycle.

4.6. View Comments

Now that all the defects are logged and submitted in inspectors profile it can be viewed in authors profile by clicking on Documents>View Comments> View (can chose either of the lists to view comments as shown in figure 4.10).

Figure 4.10. Shows author profile for viewing comments.

5. FUTURE IMPROVEMENTS

It can incorporate email facility. Any team of inspection will prefer to meet and discuss the results of inspection process. So to satisfy such needs it can also have features for users to invite for meeting sessions along with email facility. It can have user statistics to show how many defects were found by user in an hour supported with graphs for further analysis of inspection process to improve productivity of the inspector. Also, inspector profiles can have enhancements showing due date for each inspection in the 'FileList' window. Each defect log form (Fault/Error List, Error/New Fault List) can show which document is currently being inspected furthermore both fault list and new fault list forms can have drop down list with fault classes(e.g., G, MF, MP, MI, ME, AI, II, IF, WS, O) to select from while recording faults. A comments section can be added in fault list. A search capability in fault list for inspectors will also be of great help.

6. CONCLUSION

This paper was an effort to develop automated tool which supports proven techniques of finding defect in requirements. It replaces process of inspecting requirements checklist which is currently done manually using pen and paper to list all the defects found during requirements inspection. It facilitates error inspection process by assigning authors and inspectors to participate upload requirements specification document. It also provides freedom for inspectors to educate themselves with the help of inbuilt tutorials to detect defects and record their findings in three stages. Inspectors read the requirements document uploaded by authors to find fault list, using fault list by reasoning what caused those faults they find errors which is classified with the help of error taxonomy and finally use the error information to re-inspect requirements document to find the faults that might have been overlooked during the first inspection.

This tool is confined for use in requirements phase of software development. After error taxonomies have been developed and evaluated for the later phases of software development process (e.g., design, coding etc), this tool can be easily adapted to support the error-based inspection in the later phases of software development process.

REFERENCES

1. Basili, V.R., Green, S., Laitenberger, O., Lanubile, F., Shull, F., Sørumgård, S., and Zelkowitz, M.V., "The Empirical Investigation of Perspective-Based Reading." Empirical Software Engineering: An International Journal, 1(2): pp. 133-164. 1996.

2. Chillarege, R., Bhandari, I.S., Chaar, J.K., Halliday, M.J., Moebus, D.S., Ray, B.K., and Wong, M.Y., "Orthogonal defect classification-a concept for in-process measurements." IEEE Transactions on Software Engineering, 18(11): pp. 943-956. 1992.

3. Florac, W. Software Quality Measurement: A Framework for Counting Problems and Defects. Technical Reports, CMU/SEI-92- TR-22. Software Engineering Institute: 1992.

4. Sakthivel S., "A Survey of Requirements Verification Techniques," Journal of Information Technology, pp. 668-79. 1991.

5. Fagan, M. E., "Design and Code Inspections to Reduce Errors in Program Development," IBM Systems Journal, vol. 15, pp. 182-211, 1976.

6. Porter, A. A., Johnson, P.M., "Assessing Software Review Meetings : Results of a Comparative Analysis of two Experimental Studies," IEEE Transactions on Software Engineering, vol. 23, pp. 129-145, 1997.

7. Fagan, M. E., "Advances in Software Inspections," IEEE Transactions on Software Engineering, vol. 12, pp. 744-751, 1986.

8. Briand, L. C., El Emam, K., Laitenberger, O., Fussbroich,T., "Using Simulation to Build Inspection Efficiency Benchmarks for Development Projects," presented at International Conference on Software Engineering, pp. 340-449. 1998.

9. IEEE Std 610.12-1990, IEEE standard glossary of software engineering terminology. 1990.

10. Lanubile, F., Shull, F., and Basili, V.R. "Experimenting with error abstraction in requirements documents". In Proceedings of Fifth International Software Metrics Symposium, METRICS98. pp. 114-121. 1998.

11. Endres, D. Rombach, A Handbook of Software and Systems Engineering, first ed., Pearson Addison Wesley, Harlow, England, 2003.

12. S.L. Pfleeger, J.M. Atlee, Software Engineering Theory and Practice, third ed., Prentice Hall, Upper Saddle River, NJ, 2006.

13. Sommerville, Software Engineering, eighth ed., Addison Wesley, Harlow, England, 2007.

14. Card, D.N., "Learning from our mistakes with defect causal analysis." Software, IEEE. 1998. 15(1): pp. 56-63. 1998.

15. Grady, R.B., —Software Failure Analysis for High-Return Process Improvement, Hewlett-Packard Journal. 47(4): pp. 15-24. 1996.

16. Jacobs, J., Moll, J.V., Krause, P., Kusters, R., Trienekens, J., and Brombacher, A., "Exploring Defect Causes in Products Developed by Virtual Teams." Journal of Information and Software Technology. 47(6): pp. 399-410. 2005.

17. Lezak, M., Perry, D., and Stoll, D. "A Case Study in Root Cause Defect Analysis". In Proceedings of the 22nd International Conference on Software Engineering. Ireland. pp. 428-437. 2000.

18. Masuck, C., "Incorporating a Fault Categorization and Analysis Process in the Software Build Cycle." Journal of Computing Sciences in Colleges. 20(5): pp. 239 – 248. 2005

19. Mays, R.G., Jones, C.L., Holloway, G.J., and Studinski, D.P., "Experiences with Defect Prevention." IBM Systems Journal. 29(1): pp. 4 – 32. 2000.

20. Nakashima, T., Oyama, M., Hisada, H., and Ishii, N., "Analysis of Software Bug Causes and Its Prevention." Journal of Information and Software Technology. 41(15): pp. 1059-1068. 1999.

21. Chillarege, R., Bhandari, I.S., Chaar, J.K., Halliday, M.J., Moebus, D.S., Ray, B.K., and Wong, M.Y., "Orthogonal defect classification-a concept for in-process measurements." IEEE Transactions on Software Engineering. 18(11): pp. 943-956. 1992.

22. Kohn, L.T., Corrigan, J.M., and Donaldson, M.S., "To Err is Human: Building a Safer Health System. A Report of the Committee on Quality Health Care. Washington, DC. 2000.

23. Leape, L. L., "Errors in Medicine," Journal of the American Medical Association, 272(23): pp.1851-1857. 1994.

24. Norman, D.A., "Categorization of Action Slips." Psychological Review. 88: pp. 1-15. 1981.

25. Rasmussen, J., "Skills, Rules, Knowledge: Signals, Signs and Symbols and Other Distinctions in Human Performance Models." IEEE Transactions: Systems, Man, & Cybernetics. pp. 257-267. 1983.

26. Reason, J., Human Error. 1990, New York: Cambridge Press.

27. Walia, G.S. and Carver, J., —A Systematic Literature Review to Identify and Classify Requirement Errors, Journal of Information and Software Technology. 51(7). pp. 1087-1109. 2009.

28. Walia, G.S., Carver, J., and Philip, T. "Requirement Error Abstraction and Classification: An Empirical Study". In Proceedings of IEEE Symposium on Empirical Software Engineering. Brazil: ACM Press. pp. 336-345. 2006(b).

29. Walia, G., Carver, J., and Philip, T., Requirement Error Abstraction and Classification: A Control Group Replicated Study, in 18th IEEE Symposium on Software Reliability Engineering. Sweden, 2007.

30. Walia, G., Carver, J., Using Error Abstraction and Classification to Improve the Quality of Requirements: Conclusions from Family of Studies, Technical Report. 2010, NDSU, http://cs.ndsu.edu/research/ reports.htm.

31. Walia, G., Carver, J. "Evaluate the Use of Requirement Error Abstraction and Classification Method for Preventing Errors During Artifact Creation: A Feasibility Study." *Proceedings of the 21st IEEE International Symposium on Software Reliability Engineering*. November 1-4. San Jose, California, USA. pp. 81-90. 2010.

32. Kan, S.H., Basili, V.R., and Shampiro, L.N., "Software Quality: An Overview from The Perspective Of Total Quality Management." IBM Systems Journal. 33(1): pp. 4-19. 1994.

33. Walia, G.S., Empirical Validaton of Requirement Error Abstraction and Classification: A Multidisciplinary Approach, M.S Thesis, Computer Science and Engineering, Mississippi, Starkville, 2006(a).

34. Kitchenham, B. Procedures for Performing Systematic Reviews. TR/SE-0401. Department of Computer Science, Keele University and National ICT, Australia Ltd.: 2004.

35. Software Inspections. http://people.cis.ksu.edu/~hankley/d841/Fa99/Chap3.html . Date Accessed: January 31, 2011.

36. Alastair Dunsmore, "Comprehension and Visualization of Object-Oriented Code for Inspections", URL: http://www2.umassd.edu/SWPI/EFoCS/EFoCS-33-98.pdf.

37. L. Brothers and V. Sembugamoorthy and M. Muller. "ICICLE: Groupware for Code Inspection," 1990 ACM Conference on Computer Supported Cooperative Work, pages 169-181, Oct, 1990.

38. John C. Knight and E. Ann Meyers. "Phased Inspections and their Implementation," Software Engineering Notes, Vol.16, No.3, pp.29-35. July 1991.

39. John C. Knight and E. Ann Meyers. "An Improved Inspection Technique," Communications of the ACM, Vol.11, No. 11, pp.51-61. November 1993.

APPENDIX A. HARDWARE/SOFTWARE SELECTION STUDY

Programming Language

It is clear from the study results tabulated in Table A.1 all the tools are good for rapid development. The following development tools were considered:

* Poor ** Average ***Good ****Very Good *****Excellent

Table A.1. Selecting the programming language.

Criteria	Visual Basic	Visual C++	Visual C#
Ease of learning	*****	***	*****
Ease of development	*****	***	*****
Interfacing with other programs	****	***	****
Performance	***	*****	****
Functionality	****	*****	*****
Previous Knowledge and Experience	****	****	None
Rapid Development	*****	***	*****
Resource requirement	***	****	****
Network Support	****	*****	*****

From the performance perspective Visual C++ or C# looks to be the correct choice. C# was selected since it has got the most choices. There was no previous knowledge on this

language but the material available to learn was readily available and was easy to access. Another reason for selection of C# is that the predictions of the future seem that it will be a widely used language. Therefore, gaining knowledge in that area will be an added advantage for the future career.

Operating System

Since development tools selected were Visual C#, a windows based system would be required for the efficient running of the system. Currently .NET is supported only on the Microsoft windows platform. The operating system should also be easy to use, and perform well with minimum resource requirements. Familiarity and popularity are also important considerations. Considering all these factors, Windows Vista Professional was selected as the operating system.

Database

As part of the development of the tool Microsoft SQL Server 2008 was considered for the backend manipulations of user access right and table creation. Microsoft SQL Server 2008 has been considered as it is readily available in the market and easy to procure and use it in coordination with Visual Studio C# Express edition.

Hardware Requirements

Any configuration of hardware that can support Windows Vista, .NET Framework 3.5, Microsoft SQL Server 2008 is required for the implementation of the tool.