Framework for an Effective Formal Technical Review in Software Quality Assurance

Lawrence Nderu

A thesis submitted in partial fulfillment for the degree of Master of Science in Software Engineering in the Jomo Kenyatta University of Agriculture and Technology

2011

DECLARATION

This thesis is my original work and has not been presented for a degree in any other university.

Signature:…………………………………… Date:…………………………...

Lawrence Nderu

This thesis has been submitted for examination with our approval as University Supervisors.

1. Signature:…………………………… Date:…………………………

   Dr. Ronald Waweru Mwangi

   J.K.U.A.T, Kenya

2. Signature:………………………… Date:…………………………

   Dr. Stephen Kimani

   J.K.U.A.T, Kenya

DEDICATION

To Henry Gathura and Herman Ngure who have always stood by me and have been a great source of inspiration.

ACKNOWLEDGEMENTS

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. I want to thank the staff of Hospitality Software Company (HSC) Kenya, for giving me permission to do the necessary research work, make use of their departmental data and staff members.

I am deeply indebted to my supervisors Dr. Waweru Mwangi and Dr. Stephen Kimani from Jomo Kenyatta University of Agriculture and Technology whose help, stimulating suggestions and encouragement helped me in all the time of research for and writing of this thesis.

My colleagues and classmates Thuku, Eric, Noela, Wakoli and Felix supported me in my research work. I want to thank them for all their help, support, interest and valuable hints. I also want to thank Miss Esther Mumbi for all her support and her being a source of strength during my study.

Especially, I would like to give my special thanks to my brothers Henry Gathura and Herman Ngure for the support they offered to enable me to complete this work. Lastly but not the least I give thanks to Almighty God for seeing me through the tough and good times in my life; He is my only hope.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

LIST OF ABBREVIATIONS

DAO	Data Access Objects
DK	Domain Knowledge
FTR	Formal Technical Review
H.S.C.	Hospitality Systems Consultancy
NL	Natural Language
PC	Process Conformance
PM	Projects Management
SDE	Software Development Experience
SDLC	Software Development Lifecycle

ABSTRACT

Formal technical review (FTR) is an essential component of all software quality assessment, assurance and improvement techniques. However, current FTR practice leads to significant expense, clerical overhead, group process obstacles, and research methodology problems. This research aimed at looking for ways and means of making FTR more effective and less of a burden.

This research affirms that the background and experience variables of the reviewers affect the defects a reviewer is able to uncover at different phases of software development. It then presents the most important background and experience variables for a reviewer to be able to uncover certain classes of defects.

Software products are largely different, this could be due to for example, the environment of use, the effects of malfunctioning (it could be mild or fatal). This research provides a framework in which quality knowledge (lessons learned in software inspection) can be captured and reused during the inspection process. We pursue the idea of the accumulation of knowledge during reviews and establish a framework and a tool environment in which experience gained can be used by Project Managers when preparing for Formal Technical Reviews.

CHAPTER ONE

1.1 INTRODUCTION

Quality of software products has become an important area in software quality assurance, even with this importance the job of quality

Reviewing a software product is as old as programming itself. However, the first structured, measurement-based review process was Michael Fagan's five-step Inspection method (this method will be discussed in detail under literature review). The following is the overview:-

- The author presents an overview of the scope and purpose of the work product.
- Preparation: reviewers analyze the work product with the goal of understanding it thoroughly.
- Inspection meeting: the inspection team assembles and the reader paraphrases the work product. Reviewers raise issues that are subsequently recorded by the scribe.
- Rework: the author revises the work product, depending on the conclusion from the reviewers.
- Follow up: The moderator verifies the quality of rework and decides if reinspection is required [Fagan00].

The fundamental reason for measuring software and the software process is to obtain data that helps us to better control the schedule, cost, and quality of software products. It is important to be able to consistently count and measure basic entities that are directly measurable, such as size, defects, effort, and time (schedule). Consistent measurements provide data for doing the following:

- Quantitatively expressing requirements, goals, and acceptance criteria.
- Monitoring progress and anticipating problems.
- Quantifying tradeoffs used in allocating resources.
- Predicting the software attributes for schedule, cost, and quality.[Florac92]

Delaying corrections until testing and operational phases may lead to higher costs and it may be too late to improve the system significantly. Formal Technical Review (FTR) is a tool used for early prediction of fault-prone program to improve software quality. Normally FTR is conducted during each of the following phases in the software development lifecycle (SDLC).

i) Software Requirements Analysis

ii) Software Design phase

iii) Software Coding phase

iv) Software Testing phase

Formal Technical Review (FTR) is an essential component of all modern software quality assessment, assurance, and improvement techniques, and is acknowledged to be the most cost-effective form of quality improvement when practiced effectively. However, Formal Technical Review methods such as inspection are very difficult to adopt in organizations and a lot of work needs to be done to make the whole process less of a burden: first of all they introduce substantial new up-front costs, training, overhead, and group process obstacles. Sustained commitment from high-level management along with substantial resources is often necessary for successful technology transfer of FTR.

1.2 PROBLEM STATEMENT

This researcher believes that if the selection of reviewers can be made with a clear objective of the kinds of defects that individuals with various background and experience are able to uncover then the whole process of software review can be made more effective. This information can then be captured in a software product tool which will provide guidance to the project managers to perform an effective Formal Technical Review.

Much of the research that has been carried out regarding the FTR, has focused on improving the meeting and making it more effective. This line of research makes the assumption that individual inspectors already know how to inspect software artifacts on their own. Research has shown that providing inspectors with detailed techniques can improve their performance over simple ad hoc reading [Shull98]. In either case, work on the individual inspection either prior to the meeting, or in place of the meeting, has not been studied as much as the inspection meeting has. Since research has shown improvement for individuals when given a specified technique to use, there is a need for aiding the inspector, either before the inspection meeting or in the absence of the inspection meeting, to become better and more efficient at the job of detecting defects in the software artifacts.

1.3 AIMS AND OBJECTIVES

i. Investigating a useful defect classification scheme for the software development phases.
ii. Developing a list of Background and Experience variables for the reviewers.
iii. Investigating the predictive relationship between the defects uncovered and the Background and Experience of the reviewers of a software product.
iv. Developing a support tool to help project managers in selection of a team of reviewers to carry out the FTR.

This research thus, tries to answer the following question:

Do the background and Experience variables of the reviewers affect their performance during a FTR and if this is so can this information be captured in a software tool to be able to improve future reviewers.

1.4 DEFINATIONS

Software Defect can be defined as any flaw or imperfection in a software work product or software process [Brad01].

Software work product - is any artifact created as part of the software process.

Software process - is a set of activities, methods, practices and transformations that people use to develop and maintain software work products.

A defect is frequently referred to as a fault or bug.

CHAPTER TWO

2.1. METHODOLOGY

Defect classifications is subjective it is necessary to ensure that they are repeatable (i.e., that the classification is not dependent on the individual). An initial defect classification scheme in the software development lifecycle will be created. This initial classification will be delivered from literature materials and will be improved as new defect classes are discovered.

The reviewers to be involved in the questioner will have their background and experience captured as this is one of the variables that is being tested whether it affects the outcome of a review meeting. An analysis of the data collected will be carried out with the aim of trying to establish the existence of any relationship between the reviewers' background and experience and the kind of defects discovered. These results will be used to come up with a support tool for the project managers to help him/her select reviewers depending on the nature of the defects that the project manager wants uncovered.

In case the background and experience of the reviewers affects the outcome of a review, not only does the development organization have to worry about how the specific techniques and methods will positively or negatively affect their software development process, they also must have a way to evaluate the background and experience of each of their team members. The way in which these evaluations are done and the results reported by them must give the process manager some guidance on how to staff an inspection, this will be the reason why a software tool will be created to capture this details.

For this project the researcher will build a prototype to demonstrate desired functionalities and achieve objectives as stated above. To do this a back end database will be created using Microsoft SQL server 2008 with a connecting front end interface built using ASP.NET 3.5 with C#. ASP.NET 3.5 is a Microsoft development tool that allows the easy development of web applications with the incorporation of a programming language such as C# or Visual Basic. ASP.NET 3.5 is a very rich programming tool and has an in built security web control and a “membership” feature which enables the creation of roles as well as user accounts. User account data is stored in a predefined software generated database which defines the permissions of roles and users created within the system. This feature is invaluable to this project and as such the project will be exploiting the richness of this software development tool in during the build.

The Formal Technical Review Tool (FTR) created will have to be distributed so that users of the system can be able to access the software tool online. Due to this the .NET environment will be used so as to provide the .NET Remoting functionalities it also has the following advantages.

2.2. .NET FRAMEWORK ADVANTAGES

The .NET Framework offers a number of advantages to developers. Such as the following:

Consistent Programming Model - Different programming languages have different approaches for doing a task. For example, accessing data with a VB 6.0 application and a VC++ application is totally different. When using different programming languages to do a task, a disparity exists among the approach developers use to perform the task. The difference in techniques comes from how different languages interact with the underlying system that applications rely on.

With .NET, for example, accessing data with a VB .NET and a C# .NET looks very similar apart from slight syntactical differences. Both the programs need to import the System.Data namespace, both the programs establish a connection with the database and both the programs run a query and display the data on a data grid. The VB 6.0 and VC++ example mentioned in the above explains that there is more than one way to do a particular task within the same language. The .NET example explains that there's a unified means of accomplishing the same task by using the .NET Class Library, a key component of the .NET Framework. The functionality that the .NET Class Library provides is available to all .NET languages resulting in a consistent object model regardless of the programming language the developer uses.

Direct Support for Security - Developing an application that resides on a local machine and uses local resources is easy. In this scenario, security isn't an issue as all the resources are available and accessed locally. Consider an application that accesses data on a remote machine or has to perform a privileged task on behalf of a nonprivileged user. In this scenario security is much more important as the application is accessing data from a remote machine.

With .NET, the Framework enables the developer and the system administrator to specify method level security. It uses industry-standard protocols such as TCP/IP, XML, SOAP and HTTP to facilitate distributed application communications. This makes distributed computing more secure because .NET developers cooperate with network security devices instead of working around their security limitations.

Simplified Development Efforts - With classic ASP, when a developer needs to present data from a database in a Web page, he is required to write the application logic (code) and presentation logic (design) in the same file. He was required to mix the ASP code with the HTML code to get the desired result.

ASP.NET and the .NET Framework simplify development by separating the application logic and presentation logic making it easier to maintain the code. You write the design code (presentation logic) and the actual code (application logic) separately eliminating the need to mix HTML code with ASP code. ASP.NET can also handle the details of maintaining the state of the controls, such as contents in a textbox, between calls to the same ASP.NET page.

Another advantage of creating applications is debugging. Visual Studio .NET and other third party providers provide several debugging tools that simplify application development. The .NET Framework simplifies debugging with support for Runtime diagnostics. Runtime diagnostics helps you to track down bugs and also helps the developer to determine how well an application performs.

Easy Application Deployment and Maintenance - The .NET Framework makes it easy to deploy applications. In the most common form, to install an application, all you need to do is copy the application along with the components it requires into a directory on the target computer. The .NET Framework handles the details of locating and loading the components an application needs, even if several versions of the same application exist on the target computer. The .NET Framework ensures that all the components the application depends on are available on the computer before the application begins to execute.

2.3. DEVELOPMENT APPROACH

To successfully complete this project a detailed development framework will be followed. There are a number of industry standard development methods that spring to mind. Dynamic Systems Development Method (DSDM), Rapid Application Development (RAD), Structured Systems Analysis and Design Method (SSADM) are examples of standard development methods and techniques employed in industry.

These development methods are quite large and elaborate and are more suitable for use with large industry standard projects. SSADM is a more traditional and rigid approach and does not allow for iteration. With this method it is assumed that requirements would not change during the development of the project. The software product to be developed here needs to be developed in such a way that the developer is given an option of going back a step or two during the development and make necessary changes if needs be.

DSDM and RAD are more flexible methods as they both allow for iteration. These methods are user centered and require a great deal of user involvement throughout the project life cycle. DSDM stipulates that users should agree to a substantial and sustained commitment to the development process. User feedback is vital for every stage of the development cycle.

Bearing this in mind, the researcher has extracted vital bits from industry recognized frameworks and designed a framework for the project build which appropriately defines the way this software tool will be developed. The build and development of this project will be iterative and incremental in nature. A breakdown of this framework is as follows:

a) REQUIREMENTS ANALYSIS

An analysis of the needs the FTR tool should solve will be carried out. The objectives of the proposed system will be clearly defined.

b) PRIORITISE REQUIREMENTS

Requirements would be prioritized at this stage sing MoSCoW analysis. Time boxing will be employed as a time management technique.

c) REQUIREMENTS SPECIFICATION

A logical model of the system will be produced at this stage. Data models will be produced to engineer requirements of the system.

d) DESIGN AND IMPLEMENTATION

A database structure will be designed and a system will be physically produced outlining the precept of the requirements specification stage.

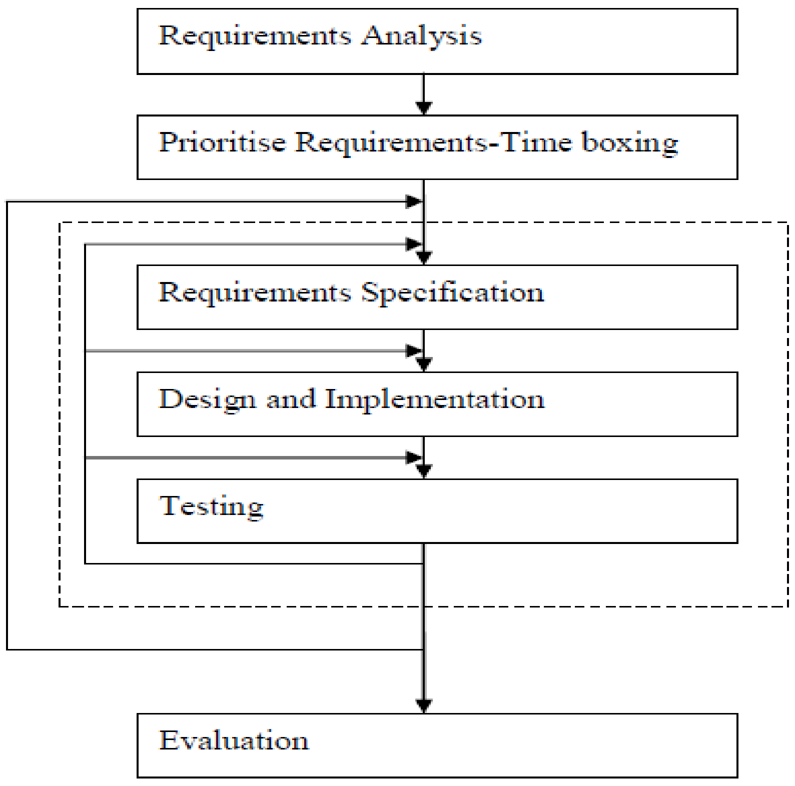
e) TESTING

A test plan is produced at this stage. The testing is iterative and is tested during the design and implementation stages.

f) EVALUATION

The system is evaluated.

2.4. FRAMEWORK DIAGRAM



Requirements Analysis
Prioritise Requirements-Time boxing
Requirements Specification
Design and Implementation
Testing
Evaluation

Fig 3.1: Development Framework Diagram

CHAPTER THREE

LITERATURE REVIEW

3.1 INTRODUCTION

The FTR (Formal Technical Review) is a software quality assurance activity with the objectives to uncover errors in function, logic or implementation for any representation of the software; to verify that the software under review meets its requirements; to ensure that the software has been represented according to predefined standards; to achieve software that is developed in a uniform manner and to make projects more manageable. FTR (Formal Technical Review) is also a learning ground for junior developers to know more about different approaches to software analysis, design and implementation. It also serves as a backup and continuity for the people who are not exposed to the software development so far. This literature review places Formal Technical Review in context as well as looking at some of reasons for carrying out Formal Technical Review.

3.2 FORMAL TECHNICAL REVIEW/INSPECTIONS

The concept of inspections is not unique to software engineering. Many fields have some type of inspection of their artifacts. The goals of these inspections can vary from setting to setting, but a general goal is ensuring that the artifact is of sufficient quality to be used by the customer(s) of that document. For example architects have their drawings inspected for feasibility before passing them along to the construction team. In the same way, software development teams have their software artifacts inspected before passing them along (to the next phase of the lifecycle). Many variations of inspections have emerged [Ackerman89].

The value of software review as a mechanism for software quality improvement has been demonstrated repeatedly. Beginning with the landmark work of Michael Fagan at IBM in 1976, structured review mechanisms such as inspection have been shown repeatedly to be an extremely effective means to find work product defects early in the software development process.

3.3 FAGAN INSPECTION MODEL

Fagan inspection refers to a structured process of trying to find defects in development documents such as programming code, specifications, designs and others during various phases of the software development process. It is named after Michael Fagan who is credited with being the inventor of Formal Software Inspections.

Fagan Inspection is a group review method used to evaluate output of a given process. It defines a process as a certain activity with a pre-specified entry and exit criteria. In every activity or operation for which entry and exit criteria are specified Fagan Inspections can be used to validate if the output of the process complies with the exit criteria specified for the process.

3.4 USAGE OF THE FAGAN INSPECTION MODEL

The software development process is a typical application of Fagan Inspection; software development process is a series of operations which will deliver a certain end product and consists of operations like requirements definition, design, coding up to testing and maintenance. As the costs to remedy a defect are up to 10-100 times less in the early operations compared to fixing a defect in the maintenance phase, it is essential to find defects as close to the point of insertion as possible [Brocklehurst92]. This is done by inspecting the output of each operation and comparing that to the output requirements, or exit-criteria of that operation.

3.5 CRITERIA

Entry criteria are the criteria or requirements which must be met to enter a specific process [Fagan, 00]. For example for Fagan inspections the high- and low-level documents must comply with specific entry-criteria before they can be used for a formal inspection process.

Exit criteria are the criteria or requirements which must be met to complete a specific process. For example for Fagan inspections the low-level document must comply with specific exit-criteria (as specified in the high-level document) before the development process can be taken to the next phase.

The exit-criteria are specified in a high-level document, which is then used as the standard to compare the operation result (low-level document) to during the inspections. Deviations of the low-level document from the requirements specified in the high-level document are called defects and can be *categorized*. The following flowchart shows the Fagan inspection stages [Eickelmann2003].

A represents the starting and ending sections

B represents the output from a section

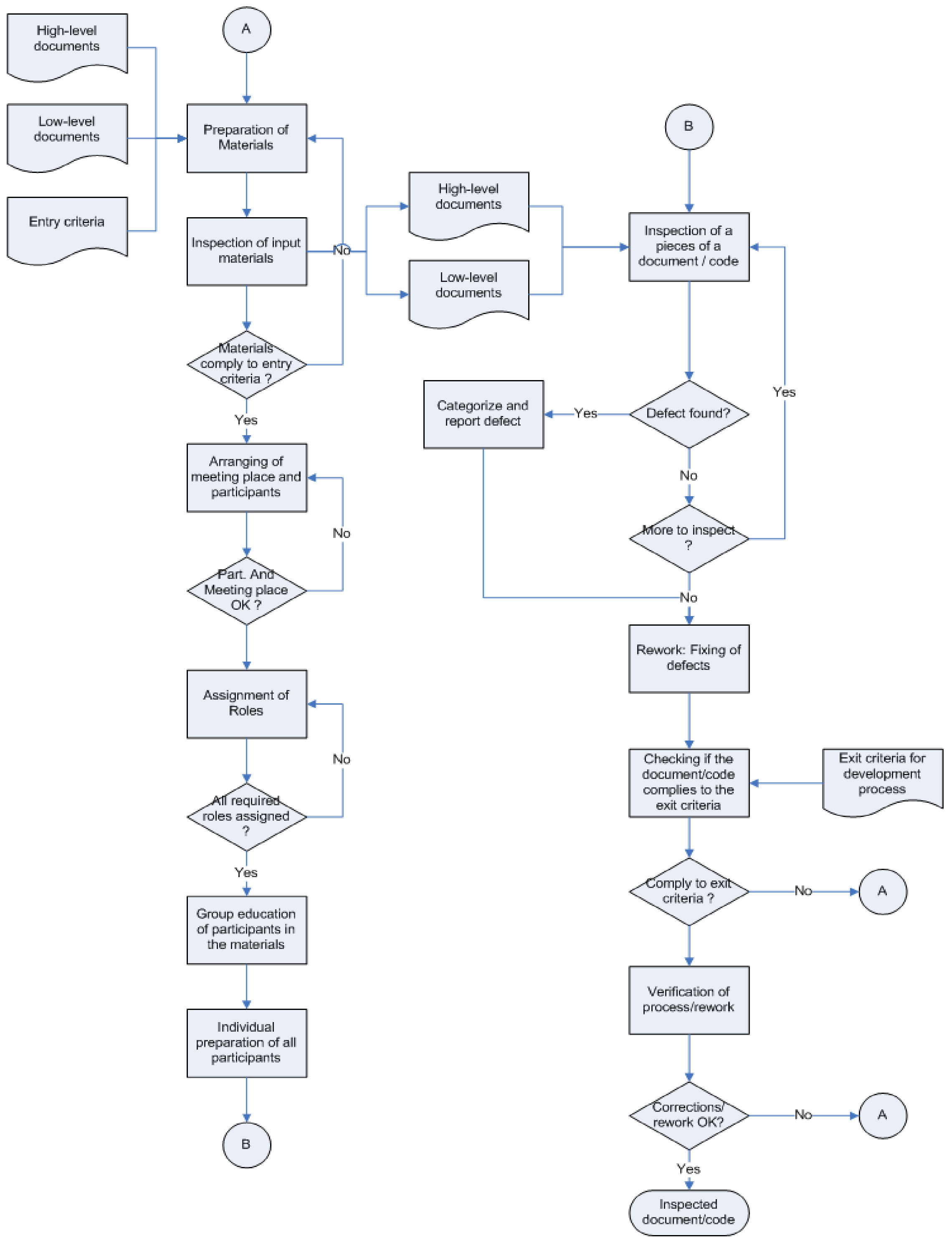

High-level documents
Low-level documents
Entry criteria
A
Preparation of Materials
Inspection of input materials
No
Materials comply to entry criteria ?
Yes
Arranging of meeting place and participants
No
Part. And Meeting place OK ?
Assignment of Roles
No
All required roles assigned ?
Yes
Group education of participants in the materials
Individual preparation of all participants
B
High-level documents
Low-level documents
B
Inspection of a pieces of a document / code
Categorize and report defect
Yes
Defect found?
Yes
No
More to inspect ?
No
Rework: Fixing of defects
Checking if the document/code complies to the exit criteria
Exit criteria for development process
Comply to exit criteria ?
No
A
Verification of process/rework
Corrections/ rework OK?
No
A
Yes
Inspected document/code

Figure 3.1 Fagan inspection model

In a typical Fagan inspection the inspection process consists of the following operations [Fagan86], [Eickelmann2003]:-

- Planning
    - Preparation of materials
    - Arranging of participants
    - Arranging of meeting place
- Overview
    - Group education of participants in the to be inspected materials
    - Assignment of roles
- Preparation
    - The participants prepare their roles
- Inspection meeting
    - Actual finding of defect
- Rework
    - Rework is the step in software inspection in which the defects found during the inspection meeting are resolved by the author, designer or programmer. On the basis of the list of defects the low-level document is corrected until the requirements in the high-level document are met.
- Follow-up
    - In the follow-up phase of software inspections all defect found in the inspection meeting should be corrected (as they have been fixed in the

rework phase). The moderator is responsible for verifying that this is indeed the case. He should verify if all defects are fixed and no new defects are inserted while trying to fix the initial defects. It is trivial that all defects are corrected as the costs of fixing them in a later phase of the project will be 10 to 100 times higher compared to the current costs

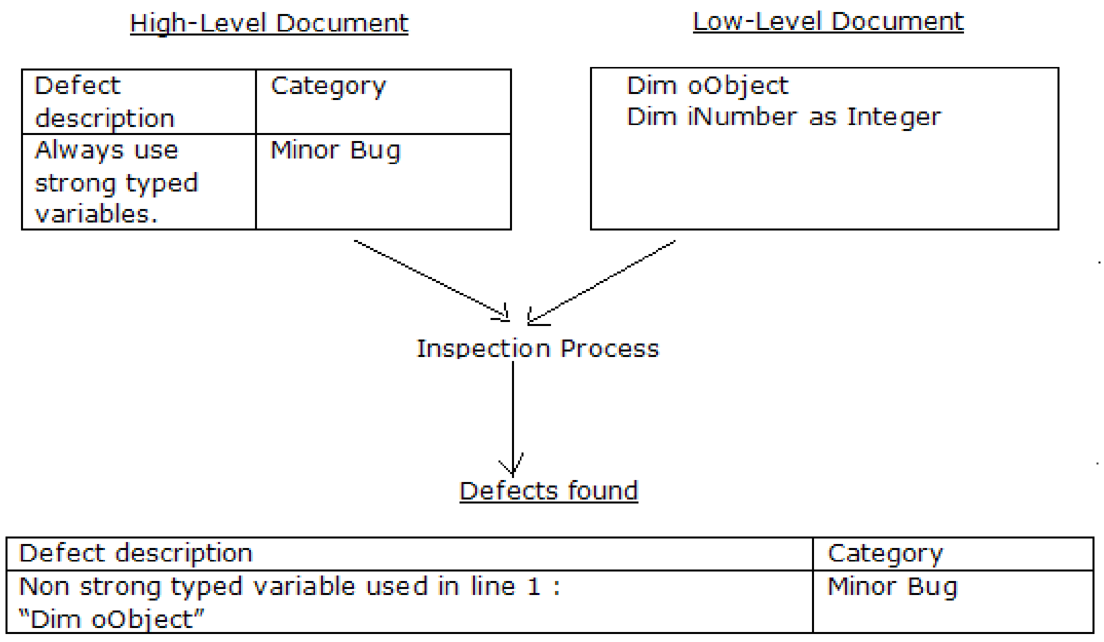
Fagan argues as follows:-
"In the process of software inspection the defects which are found are categorized in two categories: major and minor defects (often many more categories are used). The defects which are incorrect or even missing functionality or specifications can be classified as major defects: the software will not function correctly when these defects are not being solved".
By using inspections the amount of errors in the final product can significantly decrease, creating a higher quality product. In the future the team will even be able to avoid errors as the inspection sessions give them insight in the most frequently made errors. It is also possible for the team to improve on the classification of defects from the experience gained and the defects that customers of the documents will uncovered. By continuously improving the inspection process these insights can even further be used [Fagan, 2000].

Benefits have been reported by companies which have adapted the Fagan inspections model e.g. IBM indicating that 80-90% of defects can be found. A number of researchers have suggested improvements to the model with some adjustments to the original Fagan model [Doolan992].

The following example [**Figure 2.2**] shows the use of Fagan inspection model, in which the defects have been classified as either major or minor.

A Major Defect **-** This are defects such as incorrect or even missing functionality or specifications: the software will not function correctly when these defects are not solved.

A Minor Defect **-** In contrast to major defects, minor defects do not threaten the correct functioning of the software, but are mostly small errors like spelling mistakes in documents or optical issues like incorrect positioning of controls in a program interface.



Defect description	Category
Always use strong typed variables.	Minor Bug

```
Dim oObject
Dim iNumber as Integer
```

Defect description	Category
Non strong typed variable used in line 1 : "Dim oObject"	Minor Bug

Figure 2.2 Example of defect classification method

As can be seen in the high-level document for this project it is specified that in all software code produced variables should be declared as ***'strong typed'***. On the basis of this requirement the low-level document is checked for defects. Unfortunately a defect is found on line 1, as a variable is not declared 'strong typed'. The defect found is then reported in the list of defects found and categorized according to the categorizations specified in the high-level document.

As the benefits of such structured review processes (typically referred to as "Formal Technical Review" (FTR) became more visible, researchers and practitioners began to devise variations on Fagan's original method. For example, Tom Gilb developed a comprehensive inspection method with precisely defined phases, metrics, and suggested process rates for optimum defect removal effectiveness. With few exceptions, these variations never challenged a fundamental premise of Fagan's original method: that a face-to-face meeting of the entire review team is essential to the review's success. While researchers have proposed changing the manner in which reviewers prepared for the meeting, or even the manner in which the meeting was conducted, the need for a meeting was never questioned. Fagan, Gilb, and others have argued that meetings enable a kind of synergy between participants, in which defects not found by reviewers working individually suddenly come to light. They also argue that meetings educate the participants, clarify requirements, and provide milestones that facilitate progress.

Unfortunately, meetings introduce substantial costs. They require the simultaneous attendance of all participants. Their effectiveness depends on satisfying many conditions, such as adequate preparation, readiness of the work product for review, high quality moderation, and cooperative interpersonal relationships. Meeting-based review appears to add 15-20% new overhead onto development costs, and simple scheduling issues have been shown to lengthen the start-to finish interval for review by almost one third [Philip03]. The costs of meeting-based review have stimulated more recent research designed to investigate whether new review methods can be devised that minimize or eliminate the cost of meetings while preserving the remaining benefits of review (such as reduction in the cost of error corrections in a finished product). Such research has ranged from the design of computer-supported cooperative work systems that implement an asynchronous, non-meeting-based review procedure to alternative manual methods that also shift the process away from reliance on meetings.

Prior to inspections was the idea of a *walkthrough*. The walkthrough could range anywhere from a simple peer review all the way up to a formal inspection of the type discussed here. One of the problems with using *walkthrough*s in a process that is going to be improved is that normally very little data is collected because walkthroughs are less formal and applied differently in each setting [Gilb93]. Because of this the efficiency of defect detection is quite variable.

[Basili at el81] observes that all project environments and products are different in some way. Because of these differences the application of techniques and methods on different projects should be expected to vary. There are many dimensions upon which software development organizations can differ. For example, the application domain can vary. Another dimension of potential variation can be the level of risk inherent in the project. For some applications, failure may mean only mild inconveniences, while with other applications; it could mean loss of life. While there are some standard methods and practices for performing inspections, in many cases the application of those methods may need to be tailored in some way because of this variation.

The basic idea behind an inspection is that members of a software development organization review a software artifact to ensure that it possesses some level or characteristics of quality. An inspection consists of a *series of steps*. First, the author of the artifact gives the reviewers an introduction and overview of the artifact. Next, the individual inspectors review the software artifact to prepare for the technical reviewer meeting (team meeting). After the individuals have inspected the document, they meet together as a team to record the *defects* that are found. Finally, the document author is given this list of defect so that he or she may repair them in the software artifact.

Many software engineering sample sizes are small and therefore difficult to show statistical significance so qualitative data collection and analysis has to be employed in order to supplement to the more common quantitative methods. Two popular methods of qualitative data collection that have been transferred from other domains for use in software engineering are protocol analysis [Singer96], and ethnography [Shneiderman98]. These methods involve collecting data about how subjects perform processes. The data collected includes information about what the subjects did as well as what the subjects' thought processes were as they solved problems. In order to collect this type of data, researchers must employ two types of methods. The first type of methods is *retrospective*. These methods involve data collection after the process is complete, thorough post- mortems or questionnaires. This is the method that will be adapted in this study and hence the sample will come from people who have an experience in Formal Technical Review.

This method has a number of limitation one of them being that since the data is not collected until the end, there is an issue with the reliability of the information. Subjects have time to think about and formulate responses, so their response may not give a totally accurate reflection of what went on [VanSomeren94].

To be able to take care of this the **Fleiss' kappa** which is a statistical measure for assessing the reliability of agreement between a fixed number of raters when assigning categorical ratings to a number of items or classifying items will be used.

The second types of methods are *observational* and are used to collect data while the process is executing. These methods normally involve observation by the researcher of the subjects [Singer96]. The benefits of observational methods include more accurate data because the data is collected while the process is executing rather than after it is over. Also, there is no time for the subject to ‘clean up’ his answers before the researcher collects the data. On the other hand, there is the potential that the observation could cause the process to be altered. Some subjects might feel uncomfortable and act differently than they would if they were not being observed. This phenomenon is known as the Hawthorne Effect [http://www.nwlink.com/~Donclark/hrd/history/hawthorne.html]. The observational techniques are useful when the level of specificity of the data is at the level of individual steps in the procedure, rather than global information. One of the main techniques for observation is called *thinking aloud*,[VanSomeren94]. A researcher instructs the subject to recite his thinking process out loud. Then the researcher can take notes to understand what is going on.

This type of data does not lend itself to the same types of statistical analyses that can be performed on the quantitative information mentioned earlier. In analyzing qualitative data, researchers must examine the mostly textual data to look for patterns. One of the important methods of doing the analysis is *grounded theory*, which I will discuss below.

3.6 GROUNDED THEORY

The phrase "grounded theory" refers to theory that is developed inductively from a corpus of data. If done well, this means that the resulting theory at least fits one dataset perfectly. This contrasts with theory derived deductively from grand theory, without the help of data, and which could therefore turn out to fit no data at all.

Grounded theory takes a case rather than variable perspective, although the distinction is nearly impossible to draw. This means in part that the researcher takes different cases to be wholes, in which the variables interact as a unit to produce certain outcomes. A case-oriented perspective tends to assume that variables interact in complex ways, and is suspicious of simple additive models, such as ANOVA with main effects only.

Part and parcel of the case-orientation is a comparative orientation. Cases similar on many variables but with different outcomes are compared to see where the key causal differences may lie. This is based on John Stuart Mills' (1843, *A system of logic:*

Ratiocinative and Inductive) method of differences -- essentially the use of (natural) experimental design [Borgatti90]. Similarly, cases that have the same outcome are examined to see which conditions they all have in common, thereby revealing necessary causes.

The grounded theory approach, particularly the way Strauss develops it, consists of a set of steps whose careful execution is thought to "guarantee" a good theory as the outcome. Strauss would say that the quality of a theory can be evaluated by the process by which a theory is constructed. (This contrast with the scientific perspective that how you generate a theory, whether through dreams, analogies or dumb luck, is irrelevant: the quality of a theory is determined by its ability to explain new data.)

This approach to theory building is based largely on the data that has been collected in observation of the phenomenon under study [Glaser67]. *Instead of forming theories top-down* based on assumptions that the researcher has *a priori*, the theory is formed *bottom-up* systematically from the data. This method comes from the field of Sociology, and because this work is concerned with how project managers can have a way of knowing which inspectors have an edge in uncovering of defects depending on their background and experience, this technique will be useful, it is also important to note that the tool to be developed will take care of the fact that software projects are diverse.

The main idea behind this method is that as the data is analyzed, the theories are continually modified and updated to take into account each piece of data. First, after a topic of investigation has been chosen, a literature search should be performed. After this, the researcher should enter into the study with an open mind, willing to observe things that may go against his or her preconceived notions. The first case should be observed and described. Based on this information, one can begin to form theories and hypotheses. After this observation, the literature should again be searched to see if there is any other information on the specific findings from the first case that was not found in the previous literature search. The next step is to observe a second case.

While doing this, the researcher will either confirm theories and hypotheses that were discovered in the first case, or will have to modify the theories and hypothesis from the first case so that they apply to both cases. This process of reviewing new cases and modifying the hypotheses and theories to take them into account should continue until some point of confidence. This confidence could come either when one runs out of cases, or when each new case is causing very little or no change to the current theories and hypotheses [Kathy06]. At this point, the theories and hypotheses are fairly solid. This can be considered in the light of a project manager who is involved in several projects and thus carries out FTR perhaps using same reviewers or with a slight change and thus can have a database of which defects an individual with a certain kind of defects is able to uncover and continue to improve the inspection process by involving the “relevant” abilities.

Finally, [Day93] provides some rules for creating categories. This area of research applies to the creation of defect classes in this work. Each one of those classes is, in some sense, a category. So, the following rules are helpful to judge the defect classes.

- Become thoroughly familiar with the data.
- Always be sensitive to the context of the data.
- Be flexible – extend, modify and discard categories.
- Consider connections and avoid needless overlap.
- Record the criteria on which category decisions are based.
- Consider alternative ways of categorizing and interpreting the data.

These pointers will be used for evaluating and evolving both the list of experience variables as well as the defect classification schemes. [Seaman97] discusses the use of this approach when in the understanding of communication among members of a software development organization. Better and new techniques for software engineering are always developed by researchers. While these new techniques are often theoretically beneficial, in many cases little empirical support is given. In addition, it is rare to find researchers who investigate not only their new technique, but also the context in which the technique is to be used. There are many reasons for the lack of the second type of research, one being that it is very difficult to do it.

3.7 DEFECT CLASSIFICATION SCHEMES

In order to understand the types of defects that occur in software artifacts better, researchers have developed defect classification schemes. Defect classification schemes, if properly specified, can be useful for repeatability and comparisons across studies and environments [Basili84]. These schemes attempt to group the defects that occur in a particular environment into classes. Below is the discussion of work done in the area of classifying defects. The scope and specificity of the classifications described below varies greatly depending on the goals of the organization. After the related work is discussed, my approach to defect classification will be described and an initial defect classification model will be developed.

3.8 RELATED WORK IN DEFECT CLASSIFICATION

Defects classification is not a new idea. Any organization that wishes to measure their process with the intent of improving must classify the defects that are found [Basili81]. This can range from a relatively simple classification of only 2 classes, major and minor as discussed earlier, to something much more elaborate. The types of classification that an organization uses will be based on its reason for classifying the defects. The classification will also depend on the lifecycle phase in which the inspection is going to occur.

Taking into account their local environments, researchers have proposed many different defect classifications schemes. Some are said to be useful for documents produced throughout the software lifecycle. Others are directed at documents from a specific lifecycle phase. Below I will discuss already done classification of defects.

3.9 CLASSIFICATION OF DEFECTS / BUGS

There are various ways in which we can classify software defects. Below are some of the classifications:

Severity Wise:

- **Major:** A defect, which will cause an observable product failure or departure from requirements.
- **Minor:** A defect that will not cause a failure in execution of the product.
- **Fatal:** A defect that will cause the system to crash or close abruptly or affect other applications. [Robbins 08]

Work product wise:

- **SSD:** A defect from System Study document
- **FSD:** A defect from Functional Specification document
- **ADS:** A defect from Architectural Design Document
- **DDS:** A defect from Detailed Design document

- **Source code:** A defect from Source code
- **Test Plan/ Test Cases:** A defect from Test Plan/ Test Cases
- **User Documentation:** A defect from User manuals, Operating manuals

Type of Errors Wise:

- **Comments:** Inadequate/ incorrect/ misleading or missing comments in the source code
- **Computational Error:** Improper computation of the formulae / improper business validations in code.
- **Data error:** Incorrect data population / update in database
- **Database Error:** Error in the database schema/Design
- **Missing Design:** Design features/approach missed/not documented in the design document and hence does not correspond to requirements
- **Inadequate or sub optimal Design:** Design features/approach needs additional inputs for it to be complete Design features described does not provide the best approach (optimal approach) towards the solution required
- **In correct Design:** Wrong or inaccurate Design
- **Ambiguous Design:** Design feature/approach is not clear to the reviewer. Also includes ambiguous use of words or unclear design features.
- **Boundary Conditions Neglected:** Boundary conditions not addressed/incorrect

- **Interface Error:** Internal or external to application interfacing error, Incorrect handling of passing parameters, Incorrect alignment, incorrect/misplaced fields/objects, un friendly window/screen positions
- **Logic Error:** Missing or Inadequate or irrelevant or ambiguous functionality in source code
- **Message Error:** Inadequate/ incorrect/ misleading or missing error messages in source code
- **Navigation Error:** Navigation not coded correctly in source code
- **Performance Error:** An error related to performance/optimality of the code
- **Missing Requirements:** Implicit/Explicit requirements are missed/not documented during requirement phase
- **Inadequate Requirements:** Requirement needs additional inputs for to be complete
- **Incorrect Requirements:** Wrong or inaccurate requirements
- **Ambiguous Requirements:** Requirement is not clear to the reviewer. Also includes ambiguous use of words – e.g. like, such as, may be, could be, might etc.
- **Sequencing / Timing Error:** Error due to incorrect/missing consideration to timeouts and improper/missing sequencing in source code.
- **Standards:** Standards not followed like improper exception handling, use of E & D Formats and project related design/requirements/coding standards
- **System Error:** Hardware and Operating System related error, Memory leak

- **Test Plan / Cases Error:** Inadequate/ incorrect/ ambiguous or duplicate or missing - Test Plan/ Test Cases & Test Scripts, Incorrect/Incomplete test setup
- **Typographical Error:** Spelling / Grammar mistake in documents/source code
- **Variable Declaration Error:** Improper declaration / usage of variables, Type mismatch error in source code

Status Wise:

- Open
- Closed
- Deferred
- Cancelled

3.9.1 DEFECTS CLASSIFICATION SCHEMES FOR REQUIREMENTS

In this case the defects are classified according to the phases. In [Basili81] the authors describe a study of the evolution of a requirements document for the on-board operational flight program for the A-7 aircraft. This is a complex, real-time program. The defects are actual defects that were made in the evolution of the document. The authors here defined a one level classification scheme with 7 categories. These classes were used to make hypothesis about what kinds of defects that were the most common to make [Carver03].

Clerical – relatively simple problems with the document like typos.

Ambiguity – something in the document has more than one meaning.

Omission – something has been left out of the document.

Inconsistency – two parts of the document are inconsistent with one another.

Incorrect Fact – something in the document is incorrect with respect to the domain.

Information Put in Wrong Section – information included in the document was placed in the wrong section.

Implementation Fact not Included – information necessary for a proper implementation was not given.

Other – defects that do not fall into other classes.

3.9.2 DEFECTS CLASSIFICATION SCHEMES FOR CODE

There have been many studies published in which a classification of code defects has been provided. In a paper investigating testing vs. code reading [Basili87] provides two defect classification schemes for defects that can be found in code. Four programs were studied, each from a different domain. Two different languages were used for those programs, FORTRAN and Simpl-T. The four programs were:-

i) A text processor

ii) A mathematical plotting routine

iii) A numeric abstract data type

iv) A database maintainer.

The following were the defects classes:

1) **Omission –** Result from the programmer forgetting to include something in a segment of code.

2) **Commission –** Result from an incorrect segment of existing code.

And

1) **Initialization –** Improperly initializing a data structure.

2) **Computation –** Cause a calculation to evaluate the value for a variable incorrectly.

3) **Control –** Causes the wrong control path to be taken for some input.

4) **Interface –** Passing an incorrect or argument or assuming that an array passed as a parameter was padded with blanks.

5) **Data –** Result from incorrect use of a data structure.

6) **Cosmetic** - Clerical mistakes when entering a program

Khaled in a research entitled, "The Repeatability of Code Defect Classifications" [Khaled02], developed a defect classification scheme shown below

Defect Type Description and Examples of Questions

i) **Documentation**

- ***Comments, Messages***
  - Is the function described adequately at the top of the file?
  - Are variables described when declared?

- Does the function documentation describe its behavior properly?

ii) Build/Package

- ***Change management, library and version control***
  - Is there a version number defined for the file?
  - Are the correct versions of functions included in the build?

iii) Assignment

- ***Declaration, duplicate names, scope, limits***
  - Are variables initialized properly?
  - Are all library variables that capture a characteristic or state of the object defined?
  - Are all return values that are special cases (e.g., an error return) really invalid values (i.e., would never occur unless there was an error)?

iv) Interface

- ***Procedure calls and references, I/O, user formats, declarations***
  - Does the library interface correctly divide the functions into their different types?
  - Do the functions follow the proper object access rules?
  - Are the declared and expected interface signatures the same?

v) Checking

- *Error messages, inadequate checks*
  - Are all possible error conditions covered?
  - Are appropriate error messages given to the user?
  - Does the function return the <error> value in case of errors?
  - Is there checking or debugging code that is left in the function that shouldn't be there?
  - Does the function check for missing data before making a computation?
  - Are all checks for entry conditions of the function correct and complete?

vi) Data

- *Structure, content, declarations*
  - Are files opened with the right permissions?
  - Are the correct data files accessed?
  - Are there any missing variables for the object definition?
  - Are variable definitions of the right size to hold the data?

vii) Function

- *Logic, pointers, loops, recursion, computation*
  - Are all branches handled correctly?
  - Are pointers declared and used as pointers?
  - Are arithmetic expressions evaluated as specified?

viii) Memory

- ***Memory allocation, leaks***
    - Are objects instantiated before being used?
    - Do all objects register their memory usage?

ix) Environment

- ***Design, compile, test, or other support system problems***
    - Are all test cases running properly?
    - Are compile options set properly (e.g., after changing compiler version)?

x) Naming Conventions

- ***Naming of files, functions, and variables***
    - Do the function and file names follow the naming conventions for the project?
    - Do the variable names follow the naming conventions for the project?

xi) Understandability

- ***Hinder understandability***
    - Are there enough explanations of functionality or design rationale?
    - Are there any misleading variable names?

- Are the comments clear and correctly reflecting the code?

In a paper [Ackerman at al89] gives a classification scheme for defects found in requirements documents. The authors point out that for an inspection to be effective, the types of defects to be looked for must be specified. Therefore, their goal for classifying the defects is to provide the inspections with a guide to keep them on task during the inspection. While they state that inspections can be used on requirements, design, code, test plans, and test specifications, they only provide the classification for requirements.

Because the goal of this work is different from the ones presented above, a new set of defect classification schemes but which is informed by existing literature on defects classification was created. Two goals exist to this work. One is to evaluate the relationships between the defect classes and the background and experience variables. The other goal is to improve the inspection process by providing a support tool to the project managers.

3.10. DEFECTS CLASSIFICATION SCHEMES

3.10.1. DEFECTS CLASSIFICATION SCHEMES FOR REQUIREMENTS (INITIAL CLASSIFICATION)

Document Formatting issues – These defects deal with problems in the placement of requirements within the document.

Terminology – These defects occur when terminology is incorrectly or inconsistently used in such a way as to confuse a later user of the document.

States – These defects deal with system, object or data states. While it is at the requirements level, the concepts of states is still present.

Data – These defects deal with the actual data items that are to be maintained by the system. This includes inconsistent data types, validation of new data, and access restrictions to the data.

Interface and Access – These defects deal with the way that users or other systems gain access to this system. They deal both with the mechanisms of access (the interface) as well as the restrictions placed upon the access.

Other Inconsistencies – These defects deal with situations where the requirements document says contradictory things in two different parts of the document.

3.10.2 DEFECTS CLASSIFICATION SCHEMES FOR DESIGN (INITIAL CLASSIFICATION)

States – These defects occur when the states of the objects are misunderstood in some way. This includes the addition or omission of an entire state, or incorrect transitions.

Messages – These defects deal with problems involving messages that appear in the design.

Attributes – These defects deal with problems in the way that attributes have been described in the design.

Actors – These defects deal with the misuse of the actors within the design.

Constraints – These defects occur when constraints are omitted or incorrectly used.

Relationships – These defects occur when classes are incorrectly related within the design.

Class Hierarchy – These defects describe problems within the class inheritance structure.

Other – Design defects that do not fit in other categories, but are not major enough to create a new category.

3.11. CODE DEFECTS CLASSIFICATION (INITIAL CLASSIFICATION)

i) **Logic-** insufficient/incorrect errors in algorithms used.

- Wrong conditions.

ii) **Standards –** problems with coding/documentation standards.

- Indentation, alignment, layout, modularity, comments, hard-coding, and misspelling

iii) **Redundant code –** same piece of code used in many programs or in the same program

iv) **Performance –** poor processing speed: System crash because of file size, Memory problems

v) **Reusability –** inability to reuse the code

vi) **Memory management defects –** Defects such as core dump, array overflow, illegal function call, system hands, or memory overflow

vii) **Consistency –** failure to updating or delete records in the same order throughout the system

viii) **Portability –** code not independent of the platform.

3.12. TESTING DEFECTS CLASSIFICATION (INITIAL CLASSIFICATION)

Testing Tools and resources – Are the tools and resources identified and available.

Record keeping – has the record keeping mechanism being established.

Stress Testing Mechanisms – is the stress testing mechanism established?

Major identification of test phases – identification and sequencing of major test phases and the sequencing.

Consistence of the test plan with the overall project plan

3.13. INITIAL LIST OF EXPERIENCE VARIABLES

It makes sense that the background and experiences of an inspector will have an effect on the way he or she performs an inspection. This fact has been recognized by many researches that have studied inspections and most researchers take this fact into account when planning a study or experiment [Gilb93], [Sauer00]. Most of the time the researchers do not specifically address this fact, but often their observations and conclusions point to this background as a factor. The basis of this initial experience and background variables is literature which provides a basis for the creation of initial list.

i) **Domain Knowledge –** This deals with the amount of knowledge about the domain, such as banking or satellite control, which contains the problem being solved by the software being developed.

ii) **Software Development Experience –** This contains experience in whatever development phase the inspection is occurring, e.g. Requirements, Design, Code, etc., as well as with specific technologies being used in the development process, such as SCR or Object Oriented Design.

iii) **Natural Language –** This deals with how familiar an inspector is with the language the document is written in.

iv) **Experience in Project Management –** This is management of software projects.

v) **Process Conformance –** This deals with how closely the inspector follows the process that they have been given to aid in the inspection e.g. a checklist.

CHAPTER FOUR

4.1 ANALYSIS OF RESULTS

The data obtained from the ***Questionnaire*** (presented in the list of appendix) was analyzed. The objective of this section is to discuss different coefficients that can be used for evaluating agreement in defect classification for inspectors.

Data from a reliability study can be represented in a table such as Fig 5 with k defect classes. Here an expert in Formal Technical Review will independently classify the kind of background and experience that a reviewer with a certain background and experience will uncover.

	Expert B					
Expert A		Class$_1$	Class $_2$	…	Class $_K$	
	Class$_1$	P_{11}	P_{12}		P_{1k}	P_{1+}
	Class$_2$	P_{21}	P_{22}		P_{2k}	P_{2+}
	…					
	Class$_k$	P_{k1}	P_{k2}		P_{kk}	P_{k+}
		P_{+1}	P_{+2}		P_{+k}	

Table 4.1 Example table for representing *proportions* of defect classifications made by inspectors.

The method to be used to calculate the Agreement between the rates is **Fleiss' kappa** which is a statistical measure for assessing the reliability of agreement between a fixed numbers of raters when assigning categorical ratings to a number of items or classifying items. The Fleiss' kappa is an extension of the Cohen's kappa method which measures the agreement between two raters who each classify *N* items into *C* mutually exclusive categories. The first evidence of Cohen's Kappa in print can be attributed to Galton 1892 [Cohen60], [Bishop09].

The equation for κ is:

$$\kappa = \frac{\Pr(a) - \Pr(e)}{1 - \Pr(e)},$$ Where

Pr (*a*) is the relative observed agreement among raters

Pr (*e*) is the hypothetical probability of chance agreement, using the observed data to calculate the probabilities of each observer randomly saying each category.

If the raters are in complete agreement then $\kappa = 1$. If there is no agreement among the raters (other than what would be expected by chance) then $\kappa \leq 0$.

Example

Suppose that you were analyzing data related to people applying for a grant. Each grant proposal was read by two people and each reader either said "Yes" or "No" to the proposal. Suppose the data was as follows, where rows are reader A and columns are reader B:

		B	
		Yes	**No**
A	**Yes**	20	5
	No	10	15

In the notation from above we can see that the observed percentage agreement is:-

Pr (*a*) = (20+15)/50 = 0.70.

To calculate Pr (*e*) (the probability of random agreement) we note that:

- Reader A said "Yes" to 25 applicants and "No" to 25 applicants. Thus reader A said "Yes" 50% of the time.
- Reader B said "Yes" to 30 applicants and "No" to 20 applicants. Thus reader B said "Yes" 60% of the time.

Therefore the probability that both of them would say "Yes" randomly is 0.50*0.60=0.30 (Considering them to be independent) and the probability that both of them would say "No" is 0.50*0.40=0.20. Thus the overall probability of random agreement is Pr ("e") = 0.3+0.2 = 0.5.

So now applying our formula for Cohen's Kappa we get:

$$\kappa = \frac{\Pr(a) - \Pr(e)}{1 - \Pr(e)} = \frac{0.70 - 0.50}{1 - 0.50} = 0.40$$

This is a fair agreement [Brennan81].

If the results were made by chance, neither reader showing judgment the value would be zero. If the readers were in perfect agreement, the number of agreements would equal the number of trials and Kappa would be 1.

From **Table 2.1** above P_{ij} is the proportion of ratings classified in cell (i,j), P_{i+} is the total proportion for row i, and P_{+j} is the total proportion for column j:

$$Pi+ = \sum_{j=1, k} P_{ij}$$

$$P+j = \sum_{i=1, k} P_{ij}$$

The first straightforward approach to evaluating agreement is to consider the proportion of ratings upon which the two experts agree:

$$P_0 = \sum_{i=1, k} P_{ii}$$

However, this value includes agreement that could have occurred by chance. For example, if the two experts employed completely different criteria for classifying defects, then a considerable amount of observed agreement would still be expected by chance. There are different ways for evaluating extent of agreement that is expected by chance. The first assumes that chance agreement is due to the experts assigning defects uncovered by reviewers with different background and experience randomly at equal rates. In such a case chance agreement would be:

$P_e = 1/k$ ……….………………………………………………..equation i

An alternative definition of chance agreement considers that the inspectors' tendency to distribute their classifications in a certain way is a source of disagreement:

$P_e = \sum_{i=1,k} P_{i+}P_{+i}$..equation ii

The marginal proportions in the above equation are maximum likelihood estimates of the population proportions under a multinomial sampling model.

If each of the inspectors makes classifications at random according to the marginal proportions, then the above is chance agreement (derived using the multiplication rule of probability and assuming independence between the two inspectors). A general form for agreement coefficients is:

$$\text{Agreement} = (P_0 - P_e)/(1-P_e)$$

When there is complete agreement between the two inspectors, P_0 will take on the value of 1. The observed agreement that is in excess of chance agreement is given by $P_0 - P_e$ The maximum possible excess over chance agreement is $1 - P_e$.

Therefore, this type of agreement coefficient is the ratio of observed excess over chance agreement to the maximum possible excess over chance agreement. If there is complete agreement, then the agreement coefficient is 1. If observed agreement is greater than chance, then the agreement coefficient is greater than zero. If observed agreement is less than would be expected by chance, then the agreement coefficient is less than zero.

An agreement coefficient that considers chance agreement as in Equation [i], Is Bennett et al.'s S coefficient. An agreement coefficient that considers chance agreement as in Equation (ii) is Cohen's Kappa (k). Kappa has been presented as a measure of agreement in diagnosis reliability studies in many scientific fields.

Extensive use in various disciplines means that guidelines have been developed for interpreting a particular statistic. A review of the literature in various disciplines provides guidelines for interpreting Kappa, as well as interpretation guidelines for using Kappa in evaluating the reliability of software process assessments [Cohen87].

κ	**Interpretation**
< 0	No agreement
0.0 — 0.20	Slight agreement
0.21 — 0.40	Fair agreement
0.41 — 0.60	Moderate agreement
0.61 — 0.80	Substantial agreement
0.81 — 1.00	Almost perfect agreement

Table 4.2 Analysis of K value

Fleiss' kappa which as stated above is an extension of the Cohen’s Kappa value

4.2 FLEISS' KAPPA

Fleiss' kappa is a generalization of Scott's pi statistic, a statistical measure of inter-rater reliability. It is also related to Cohen's kappa statistic. Whereas Scott's pi and Cohen's kappa work for only two raters, Fleiss' kappa works for any number of raters giving categorical ratings, to a fixed number of items. It can be interpreted as expressing the extent to which the observed amount of agreement among raters exceeds what would be expected if all raters made their ratings completely randomly.

Agreement can be thought of as follows, if a fixed number of people assign numerical ratings to a number of items then the kappa will give a measure for how consistent the ratings are. The kappa,κ, can be defined as,

$$\kappa = \frac{\bar{P} - \bar{P}_e}{1 - \bar{P}_e}$$

The factor $1 - \bar{P}_e$ gives the degree of agreement that is attainable above chance and $\bar{P} - \bar{P}_e$ gives the degree of agreement actually achieved above chance. If the raters are in complete agreement then:-

$\kappa = 1$. If there is no agreement among the raters (other than what would be expected by chance) then, $\kappa \leq 0$.

In this study the Fleiss' kappa was used to find the degree of agreement between the Expect reviewers.

For each of the defects the analysis was done as follows below:

Let *N* be the total number of subjects (the background and experience variables), let *n* be the number of raters per subject, and let *k* be the number of categories into which assignments are made. The subjects are indexed by *i* = 1, 2... *N* and the categories are indexed by *j* = 1, ... *k*. Let n_{ij} represent the number of raters who assigned the *i*-th subject to the *j*-th category.

First calculate p_j, the proportion of all assignments which were to the *j*-th category:

$$p_j = \frac{1}{Nn} \sum_{i=1}^{N} n_{ij}, \qquad 1 = \frac{1}{n} \sum_{j=1}^{k} n_{ij}$$

Now calculate P_i, the extent to which raters agree for the *i*-th subject:

$$P_i = \frac{1}{n(n-1)} \sum_{j=1}^{k} n_{ij}(n_{ij} - 1)$$

$$= \frac{1}{n(n-1)} \sum_{j=1}^{k} (n_{ij}^2 - n_{ij})$$

$$= \frac{1}{n(n-1)} [(\sum_{j=1}^{k} n_{ij}^2) - (n)]$$

Now compute $\bar{P}$, the mean of the P_i's, and $\bar{P}_e$ which go into the formula for κ:

$$\bar{P} = \frac{1}{N}\sum_{i=1}^{N} P_i$$

$$= \frac{1}{Nn(n-1)}(\sum_{i=1}^{N}\sum_{j=1}^{k} n_{ij}^2 - Nn)$$

$$\bar{P}_e = \sum_{j=1}^{k} p_j^2$$

Two variations of *kappa* are provided: Siegel and Castellan's (1988) fixed-marginal multirater *kappa* and a multirater variation (Randolph, 2005). Brennan and Prediger suggest using free-marginal *kappa* when raters are not forced to assign a certain number of cases to each category and using fixed-marginal *kappa* when they are, thus in this study the Free- marginal Kappa was used since the raters were not forced to assign a certain number of cases to each category. The sample used was from a Hospitality software development company called Hospitality Systems Consultancy (H.S.C.) located in Nairobi, A sample of 10 experts in software development were used (Ten was the number of experts who could be able to participate in the exercise).

The mode of data collection was a combination of interview and questioner. Data was collected for three days just immediately after a software product developed by the company was released to the market and a number of reviews were carried out on the product.

4.3 CALCULATION OF K VALUE FOR DIFFERENT PHASES

NOTATION

DK – Amount of knowledge about the domain containing the problem being solved.

SDE - Experience in whatever development phase the inspection is occurring at**.**

NL - Familiarity with the documents language.

PM – Experience in software projects management

PC- How closely the inspector uses the checklist or the process provided.

The background and experience variable with the highest value is picked.

4.3.1 REQUIREMENTS PHASE

n = 10 – Number of Expect raters

N = 5 – Background and Experience variables

k = 5 – Categories e.g. Agree, Strongly Agree etc

i) Document Formatting Issues

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	4	2	0	0	4
SDE	6	0	1	0	3
NL	2	8	0	0	0
PM	4	4	1	0	1
PC	9	0	1	0	0

Table 5.1.1 Document Formatting Issues Results

Percent of overall agreement Po: 0.480000

Fixed-marginal kappa: 0.190535

Free-marginal kappa: 0.35000

A k value of 0.35 is Fair Agreement and thus the data is fairly reliable, hence

NL and PC are highly important for one to uncover Document formatting issues under requirements.

ii) **Terminology –** These defects occur when terminology is incorrectly or inconsistently

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	7	1	0	2	0
SDE	5	4	0	0	1
NL	9	1	0	0	0
PM	6	2	1	0	1
PC	7	1	0	1	1

Table 5.1.2 Terminology Issues Results

Percent of overall agreement Po: 0.493333

Fixed-marginal kappa: -0.0182215

Free-marginal kappa: 0.366666

A k value of 0.366 is Fair Agreement and thus the data is fairly reliable, hence

NL and SDE are the most important factors for one to be able to uncover Terminology issues under requirements

iii) **States** – These defects deal with system, object or data states.

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	4	1	3	2	0
SDE	6	4	0	0	0
NL	2	1	0	1	6
PM	6	4	0	0	0
PC	5	1	2	1	1

Table 5.1.3 States Issues Results

Percent of overall agreement Po: 0.351111

Fixed-marginal kappa: 0.0782827

Free-marginal kappa: 0.188889

A k value of 0.188889 is Slight Agreement and thus the data is slightly reliable, hence PM and SDE are the highly most important factors for Terminology issues.

iv) **Data** – These defects deal with the actual data items that are to be maintained by the system. This includes inconsistent data types, validation of new data, and access restrictions to the data.

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	1	4	2	1	2
SDE	4	2	0	0	4
NL	6	0	0	4	0
PM	4	4	1	1	0
PC	8	2	0	0	0

Table 5.1.4 Data Issues Results

Percent of overall agreement Po:	0.368889
Fixed-marginal kappa:	0.0963474
Free-marginal kappa:	0.211111

A k value of 0.211 is a Fair Agreement and thus the data is fairly reliable, hence PC and PM are the most important factors for one to be able to uncover data related defects under requirements.

v) **Interface and Access –** These defects deal with the way that users or other systems gain access to this system. They deal both with the mechanisms of access (the interface) as well as the restrictions placed upon the access.

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	8	2	0	0	0
SDE	9	1	0	0	0
NL	6	0	0	4	0
PM	4	4	1	1	0
PC	8	2	0	0	0

Table 5.1.5 Interface and Access Issues Results

Percent of overall agreement Po: 0.564444

Fixed-marginal kappa: 0.0677312

Free-marginal kappa: 0.455555

A k value of 0.4555 is a Moderate agreement and thus the data is moderately reliable, hence PC, DK and SDE are the most important factors for one to be able to uncover interface and access related defects under requirements.

vi) **Other Inconsistencies –** These defects deal with situations where the requirements document says contradictory things in two different parts of the document.

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	0	0	3	2	5
SDE	2	2	1	2	3
NL	2	2	0	4	2
PM	2	2	3	2	1
PC	3	2	1	2	2

Table 5.1.6 Other Inconsistencies Issues Results

Percent of overall agreement Po:	0.182222
Fixed-marginal kappa:	-0.0335920
Free-marginal kappa:	-0.0222225

A k value of -0.022 shows no agreement and thus the data is not reliable.

4.3.2 DESIGN PHASE

i) **States –** These defects occur when the states of the objects are misunderstood in some way. This includes the addition or omission of an entire state, or incorrect transitions.

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	1	1	6	0	2
SDE	7	3	0	0	0
NL	0	1	4	2	3
PM	5	3	0	0	2
PC	9	1	0	0	0

Table 5.2.1 Other Inconsistencies Issues Results

Percent of overall agreement Po:	0.444444
Fixed-marginal kappa:	0.220600
Free-marginal kappa:	0.305555

A k value of 0.30555 is a fair agreement and thus the data is fairly reliable, hence PC, PM and SDE are the most important factors for one to be able to uncover state related defects under design.

ii) **Messages –** These defects deal with problems involving messages that appear in the design.

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	1	1	2	5	1
SDE	7	2	1	0	0
NL	0	2	0	2	6
PM	8	0	1	0	1
PC	6	3	0	1	0

Table 5.2.2 Messages Issues Results

Percent of overall agreement Po:	0.426666
Fixed-marginal kappa:	0.207226
Free-marginal kappa:	0.283333

A k value of 0.283333 is a fair agreement and thus the data is fairly reliable, hence PC, PM and SDE are the most important factors for one to be able to uncover messages defects under design phase.

iii) **Attributes –** These defects deal with problems in the way that attributes have been described in the design

.

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	3	2	3	0	2
SDE	9	0	0	0	1
NL	2	2	0	3	3
PM	5	3	0	1	1
PC	7	2	0	1	0

Table 5.2.3 Attributes Issues Results

Percent of overall agreement Po:	0.386666
Fixed-marginal kappa:	0.0763042
Free-marginal kappa:	0.233333

A k value of 0.233333 is a fair agreement and thus the data is fairly reliable, hence PC, PM and SDE are the most important factors for one to be able to uncover attributes defects under design phase.

iv) **Actors –** These defects deal with the misuse of the actors within the design.

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	3	2	3	0	2
SDE	8	2	0	0	0
NL	2	2	0	3	3
PM	5	3	0	1	1
PC	6	4	0	0	0

Table 5.2.4 Actors Issues Results

Percent of overall agreement Po:	0.351111
Fixed-marginal kappa:	0.0423716
Free-marginal kappa:	0.188889

A k value of 0.188889 is a slight agreement and thus the data is slightly reliable, hence PC, PM and SDE are the most important factors for one to be able to uncover actors defects under design phase.

v) **Constraints –** These defects occur when constraints are omitted or incorrectly used.

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	7	2	1	0	0
SDE	9	1	0	0	0
NL	2	2	0	3	3
PM	5	3	0	1	1
PC	4	4	0	2	0

Table 5.2.5 Constraints Issues Results

Percent of overall agreement Po:	0.408888
Fixed-marginal kappa:	0.0611309
Free-marginal kappa:	0.26111

A k value of 0.26111 is a fail agreement and thus the data is fairly reliable, hence DK, PM and SDE are the most important factors for one to be able to uncover Constraints defects under design phase.

vi) **Relationships –** These defects occur when classes are incorrectly related within the design

.

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	0	2	1	0	7
SDE	1	1	0	0	8
NL	2	1	0	3	4
PM	0	1	0	0	9
PC	2	0	0	1	7

Table 5.2.6 Relationships Issues Results

Percent of overall agreement Po: 0.524444

Fixed-marginal kappa: 0.0158195

Free-marginal kappa: 0.405555

In this case the data is reliable since k=0.405555, but shows that the experts do not know the which factors has influence on uncovering of relationships defects under design phase (may be this class of defects was not well placed)

vii) **Class Hierarchy –** These defects describe problems within the class inheritance structure.

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	2	2	0	0	6
SDE	7	3	0	0	0
NL	2	1	0	3	4
PM	2	4	0	3	1
PC	9	1	0	0	0

Table 5.2.7 Class Hierarchy Issues Results

Percent of overall agreement Po:	0.431111
Fixed-marginal kappa:	0.181690
Free-marginal kappa:	0.288889

A k value of 0.288889 is a fail agreement and thus the data is fairly reliable, hence PC and SDE are the most important factors for one to be able to uncover Constraints defects under design phase.

viii) **Other –** Design defects that do not fit in other categories, but are not major enough to create a new category.

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	0	0	0	0	10
SDE	0	0	0	0	10
NL	0	2	0	1	7
PM	0	0	0	0	10
PC	1	1	0	0	8

Table 5.2.8 Other Issues Results

Percent of overall agreement Po:	0.822221
Fixed-marginal kappa:	0.0421390
Free-marginal kappa:	0.777776

A k value of 0.777776 is a Substantial agreement and thus the data is substantially reliable.

4.3.3 CODING PHASE

i) **Logic-** insufficient/incorrect errors in algorithms used. Wrong conditions.

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	0	0	0	0	10
SDE	8	2	0	0	0
NL	0	2	0	1	7
PM	2	3	3	0	2
PC	9	0	0	0	1

Table 5.3.1 Logic Issues Results

Percent of overall agreement Po:	0.622222
Fixed-marginal kappa:	0.437830
Free-marginal kappa:	0.527778

A k value of 0.527778 is a Moderate agreement and thus the data is moderately reliable, hence PC and SDE are the most important factors. All the experts do not know whether the DK affects the Logic defects uncovering under coding phase.

ii) **Standards –** problems with coding/documentation standards. Indentation, alignment, layout, modularity, comments, hard-coding, and misspelling

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	4	0	2	0	4
SDE	8	2	0	0	0
NL	5	4	0	1	0
PM	2	3	3	0	2
PC	9	0	0	0	1

Table 5.3.2 Standards Issues Results

Percent of overall agreement Po:	0.453333
Fixed-marginal kappa:	0.123931
Free-marginal kappa:	0.316666

A k value of 0.316666 is a Fair agreement and thus the data is fairly reliable, hence PC, NL and SDE are the most important factors for an expert reviewer to be able to uncover standards defects in the coding phase.

iii) **Redundant code –** same piece of code used in many programs or in the same program

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	4	0	2	0	4
SDE	10	0	0	0	0
NL	1	2	6	1	0
PM	2	3	3	0	2
PC	9	1	0	0	0

Table 5.3.3 Redundant code Issues Results

Percent of overall agreement Po:	0.524444
Fixed-marginal kappa:	0.270620
Free-marginal kappa:	0.405555

A k value of 0.405555 is a Moderate agreement and thus the data is moderately reliable, hence SDE and PC are the most important factors for a reviewer to be able to uncover redundant code defects under coding phase.

iv) **Performance –** poor processing speed: System crash because of file size, Memory problems

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	9	1	0	0	0
SDE	8	2	0	0	0
NL	1	1	6	2	0
PM	4	3	0	0	3
PC	8	1	0	0	1

Table 5.3.4 Performance Issues Results

Percent of overall agreement Po: 0.537777

Fixed-marginal kappa: 0.219218

Free-marginal kappa: 0.422221

A k value of 0.422221 is a Moderate agreement and thus the data is moderately reliable, hence SDE, DK and PC are the most important factors for a reviewer to be able to uncover redundant code defects under coding phase.

v) **Reusability –** inability to reuse the code

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	3	2	1	4	0
SDE	9	1	0	0	0
NL	5	0	4	0	1
PM	4	1	0	5	0
PC	10	0	0	0	0

Table 5.3.5 Reusability Issues Results

Percent of overall agreement Po: 0.546666

Fixed-marginal kappa: 0.199622

Free-marginal kappa: 0.433333

A k value of 0.43333 is a Moderate agreement and thus the data is moderately reliable, hence SDE and PC are the most important factors for a reviewer to be able to uncover Reusability defects under coding phase.

vi) **Memory management defects –** Defects such as core dump, array overflow, illegal function call, system hands, or memory overflow

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	3	2	1	4	0
SDE	9	1	0	0	0
NL	2	3	1	3	1
PM	9	1	0	0	0
PC	10	0	0	0	0

Table 5.3.6 Memory management defects Issues Results

Percent of overall agreement Po:	0.595555
Fixed-marginal kappa:	0.226978
Free-marginal kappa:	0.494444

A k value of 0.494444 is a Moderate agreement and thus the data is moderately reliable, hence SDE, PM and PC are the most important factors for a reviewer to be able to uncover Memory management defects under coding phase.

vii) **Consistency –** failure to updating or delete records in the same order throughout the system

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	3	2	2	3	0
SDE	10	0	0	0	0
NL	5	0	1	3	1
PM	9	1	0	0	0
PC	8	2	0	0	0

Table 5.3.7 Consistency Issues Results

Percent of overall agreement Po:	0.582222
Fixed-marginal kappa:	0.132521
Free-marginal kappa:	0.477778

A k value of 0.477778 is a Moderate agreement and thus the data is moderately reliable, hence SDE, PM and PC are the most important factors for a reviewer to be able to uncover Consistency defects under coding phase.

viii) **Portability –** code not independent of the platform.

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	3	1	4	2	0
SDE	10	0	0	0	0
NL	0	2	1	3	4
PM	2	1	1	4	2
PC	4	3	0	0	3

Table 5.3.8 Portability Issues Results

Percent of overall agreement Po: 0.377777

Fixed-marginal kappa: 0.177824

Free-marginal kappa: 0.222221

A k value of 0.222221 is a Fair agreement and thus the data is fairly reliable, hence SDE is the most important factors for a reviewer to be able to uncover Portability defects under coding phase.

4.3.4 TESTING PHASE

i) **Testing Tools and resources –** Are the tools and resources identified and available.

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	9	1	0	0	0
SDE	10	0	0	0	0
NL	0	3	4	1	2
PM	4	4	0	0	2
PC	7	3	0	0	0

Table 5.4.1 Testing Tools and resources Issues Results

Percent of overall agreement Po: 0.568888

Fixed-marginal kappa: 0.254647

Free-marginal kappa: 0.46111

A k value of 0.46111 is a moderate agreement and thus the data is moderately reliable, hence DK, SDE and PC are the most important factors for a reviewer to be able to uncover Testing Tools and resources defects under testing phase.

ii) **Record keeping –** has the record keeping mechanism being established.

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	4	4	0	2	0
SDE	7	1	0	0	2
NL	0	2	1	2	5
PM	4	4	0	1	1
PC	6	3	0	0	1

Table 5.4.2 Record keeping Issues Results

Percent of overall agreement Po:	0.342222
Fixed-marginal kappa:	0.0635279
Free-marginal kappa:	0.177778

A k value of 0.177778 is a slight agreement and thus the data is slightly reliable, hence DK, SDE, PM and PC are the most important factors for a reviewer to be able to uncover Record keeping defects under testing phase.

iii) **Stress Testing Mechanisms –** is the stress testing mechanism established?

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	1	8	0	0	1
SDE	10	0	0	0	0
NL	0	3	2	2	3
PM	10	0	0	0	0
PC	9	1	0	0	0

Table 5.4.3 Stress Testing Mechanisms Issues Results

Percent of overall agreement Po:	0.719999
Fixed-marginal kappa:	0.511171
Free-marginal kappa:	0.649999

A k value of 0.649999 is a Substantial agreement and thus the data is substantially reliable, hence DK, SDE, PM and PC are the most important factors for a reviewer to be able to uncover Stress Testing Mechanisms defects under testing phase.

iv) **Major identification of test phases –** identification and sequencing of major test phases and the sequencing.

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	10	0	0	0	0
SDE	10	0	0	0	0
NL	2	3	2	0	3
PM	10	0	0	0	0
PC	8	2	0	0	0

Table 5.4.4 Major identification of test phases Issues Results

Percent of overall agreement Po:	0.764444
Fixed-marginal kappa:	0.316833
Free-marginal kappa:	0.705555

A k value of 0.705555 is a Substantial agreement and thus the data is substantially reliable, hence DK, SDE, PM and PC are the most important factors for a reviewer to be able to uncover Major identification of test phases defects under testing phase.

v) **Consistence** of the test plan with the overall project plan

	Strongly Agree	Agree	Disagree	Strongly Disagree	Do not Know
DK	9	1	0	0	0
SDE	10	0	0	0	0
NL	0	1	4	4	1
PM	10	0	0	0	0
PC	8	1	0	0	1

Table 5.4.5 Consistence Issues Results

Percent of overall agreement Po:	0.737777
Fixed-marginal kappa:	0.396356
Free-marginal kappa:	0.672221

A k value of 0.737777 is a Substantial agreement and thus the data is substantially reliable, hence DK, SDE, PM and PC are the most important factors for a reviewer to be able to uncover Consistence defects under testing phase

4.4 SUMMARY OF THE RESULTS

NOTATION

√ - Shows a very important Factors

a) **Requirements Phase**

Defects	Background and Experience variables				
	DK	SDE	NL	PM	PC
Document Formatting Issues			√		√
Terminology		√	√		
States		√		√	
Data				√	√
Interface and Access	√	√			√
Other Inconsistencies	No Agreement				

Table 5.5.1 Summary of Requirements Phase Results

b) **Design Phase**

Defects	Background and Experience variables				
	DK	SDE	NL	PM	PC
States		√		√	√
Messages		√		√	√
Attributes		√		√	√
Actors		√		√	√
Constraints	√	√		√	
Relationships	No Agreement				
Class Hierarchy		√			√
Other	No Agreement				

Table 5.5.2 Summary of Design Phase Results

c) **Coding Phase**

Defects	Background and Experience variables				
	DK	SDE	NL	PM	PC
Logic		√			√
Standards		√	√		√
Redundant code		√			√
Performance	√	√			√
Reusability		√			√
Memory management defects		√		√	√
Consistency		√		√	√
Portability		√			

Table 5.5.3 Summary of Coding Phase Results

d) **Testing Phase**

Defects	Background and Experience variables				
	DK	SDE	NL	PM	PC
Testing Tools and resources	√	√			√
Record keeping	√	√		√	√
Stress Testing Mechanisms	√	√		√	√
Major identification of test phases	√	√		√	√
Consistence	√	√		√	√

Table 5.5.3 Summary of Testing Phase Results

CHAPTER FIVE

DETAILED SYSTEM DEVELOPMENT DOCUMENTATION OF THE FORMAL TECHNICAL REVIEW SUPPORT TOOL

5.1 REQUIREMENTS ANALYSIS

i) Users/ Roles

a) Administrator – This is a person who is the overall administrator of the system

His/her responsibilities are:

- Setting up the projects
- Input personal identification details of the reviewer.
- Input experience and background details of the reviewer.
- The number of defects uncovered under each class.
- Get recommendations on who would help him uncovered certain kinds/kind of defects depending on the phase of the lifecycle.
- Edit the above records.

b) **Project Manager – This is the manager of a specific project, his/her functions are**

   - He/she should be able to view the projects that he/she is assigned.

c) **Reviewer – He/she is an individual with the ability to uncovered defects in a software product.**

   - Based on his/her ability he can be able to view the projects that he/she is assigned to depending on his/her ability to uncover defects.

d) **Customer - This is the individual or company who initiated a project(s), his duties includes**

   - Viewing project defects and other project-related information for projects that he/she has initiated.

ii) Requirements Overview of the proposed system

- One system – serving a number of geographically dispersed users, this introduces the concept of distributed system.
- Different types of users who require different functionality from the system.
- The system needs to have some kind of data layer that is information needs to be stored.

- The system proposed will be a distributed system and hence the assumption that the system is network – based.

5.2 THE SUGGESTED SOLUTION

- The database to be built using Microsoft SQL Server 2008, including SQL Server Management Studio
- The main system will be built using the .NET Framework and C# programming language.
- The main system will also be implemented using asp.net to demonstrate its online accessibility and also its functionality will be implemented using a web service to demonstrate its distributed nature.
- For the UML diagrams argoUML tool will be used for design.
- To design the user interface Mockup screens will be used.
- The administrator interface will be tightly coupled to the system and will be built using .NET Remoting
- A website will be built to help in remote accessibility of the system.

Use Case Diagram

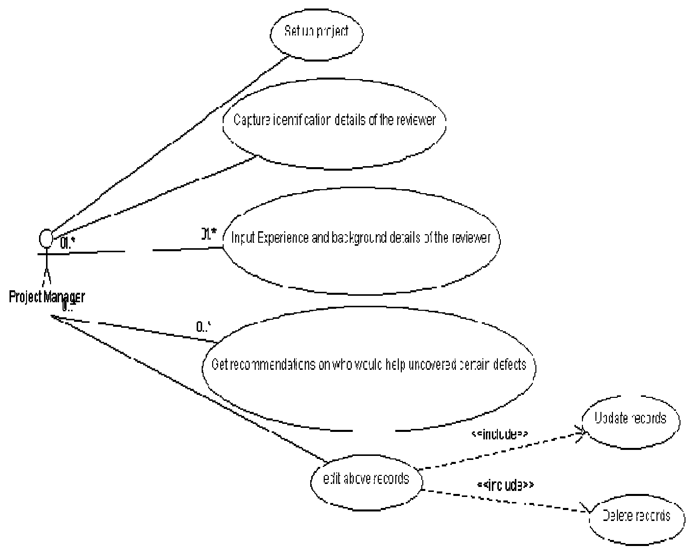

Set up project
Capture identification details of the reviewer
0..*
0..*
Input Experience and background details of the reviewer
Project Manager
0..*
0..*
Get recommendations on who would help uncovered certain defects
Update records
<<include>>
edit above records
<<include>>
Delete records

Fig 5.1 Use Case diagram for the Administrator duties

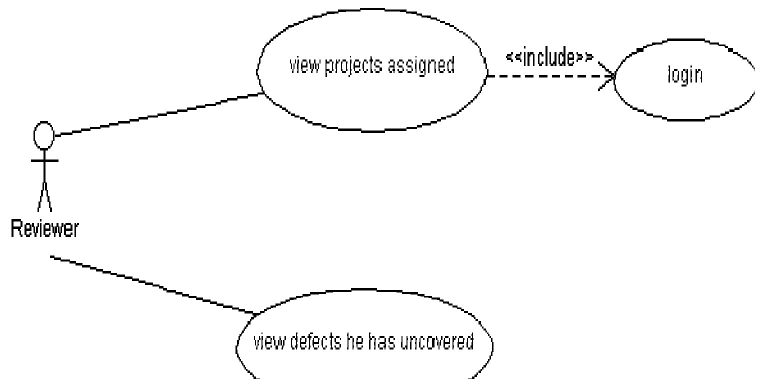


Fig 5.2 Use Case diagram for the reviewer duties

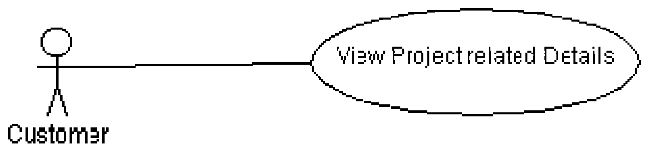


Fig 5.3 Use Case diagram for the customer duties

5.3 NON-FUNCTIONAL REQUIREMENTS

- Testability
- Maintainability
- Extensibility
- Scalability

5.4 SYSTEM DESIGN

i) System Architecture

The main system will be developed using the .NET platform since it provides a standardized set of services. “It’s just like Windows, except distributed over the Internet. It exports a common interface so that its programs can be run on any system that supports .NET” [Mark03]. The .Net Framework provides a very good environment to develop networked applications and Web Services programming API and unified language-independent development framework [Simon08].

In N-Tier architectures there is a logical separation of presentation, business and data into separate layers

- Data Tier – manages the data
- Presentation Tier – controls what a user sees and can do with the system
- Business Tier (middle tier) – controls everything else (the business logic)

The business tier contains the core functionality of the system

- ✓ Business rules
- ✓ Work flows

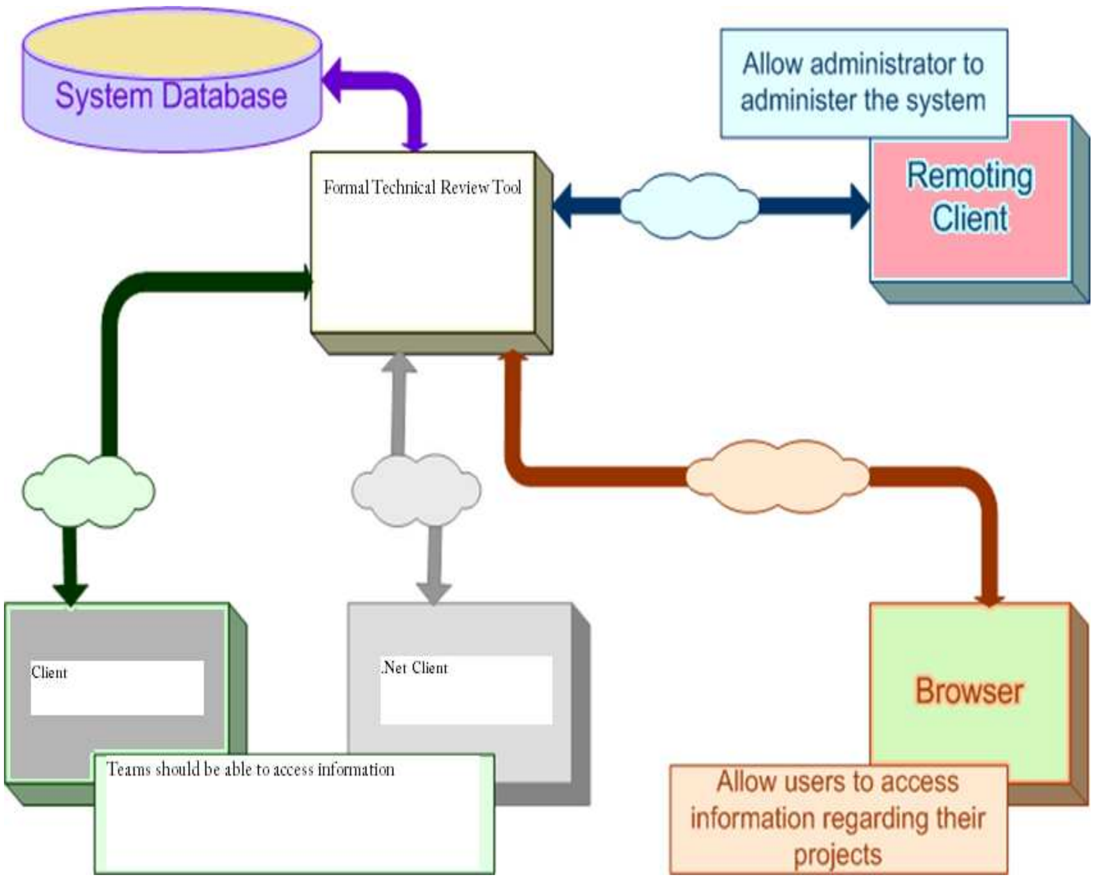
It provides controlled access to data and enables validation and processing of data input, the business tier will be defined using classes. The collection (library) of classes representing the business tier will be deployed as a component i.e. DLL

The component developed will consist of a collection of classes developed to fulfill certain specification

- It can be re-used

- It should encapsulate all its behavior
- It must provide an interface to allow it to be accessed by a client

The proposed system illustration is shown below.



System Database
Allow administrator to administer the system
Formal Technical Review Tool
Remoting Client
Client
.Net Client
Browser
Teams should be able to access information
Allow users to access information regarding their projects

Fig 5.4: The Architecture for the System

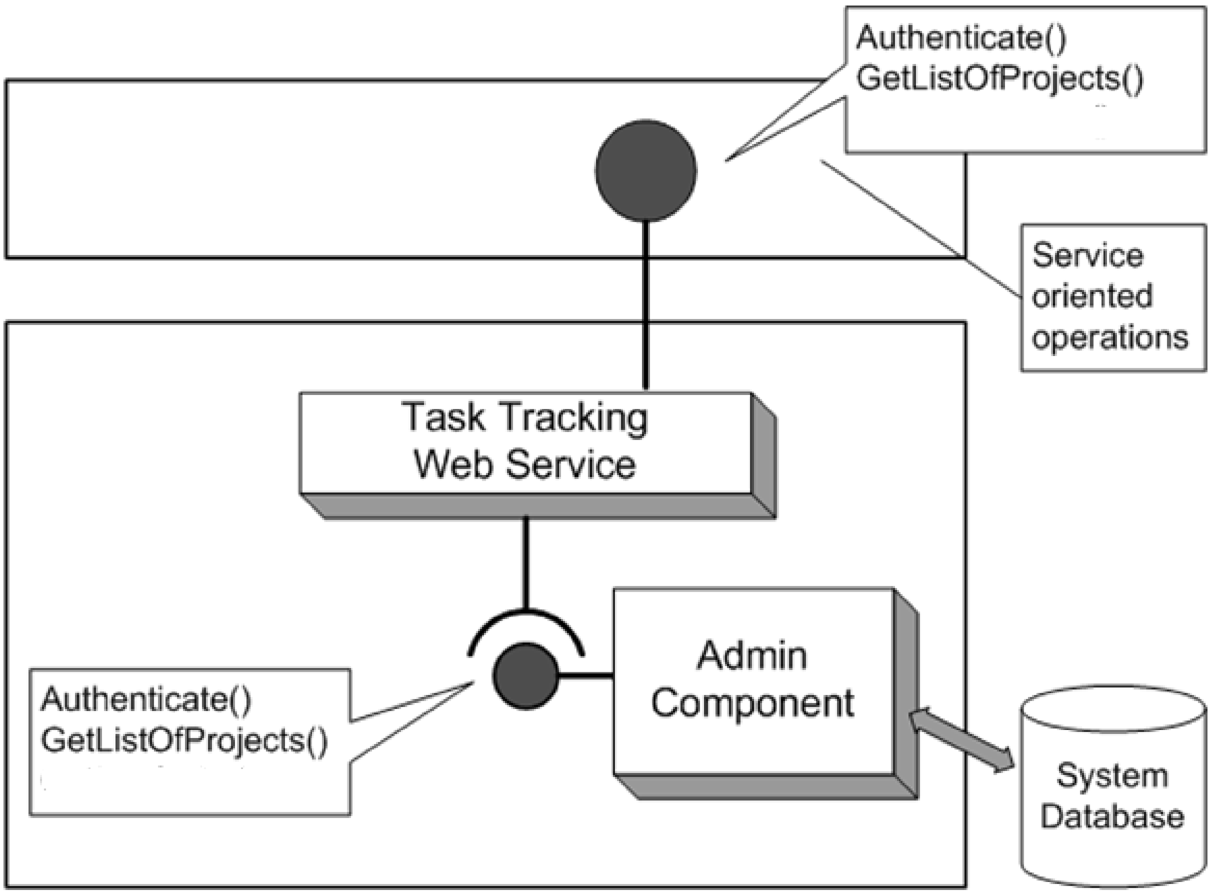


Fig 5.5 The Architecture of the System with an illustration of the methods available

5.5 THE DATABASE DESIGN

A database has been designed in order to support the Formal Technical Review Support Tool; it provides the persistence tier for the system. The following are the entities identified:

Manager

Administrator

Reviewer

ReviewerDetails

Customer

Project

Phases

Defect

ReviewAssignments.

The Database Design is as shown below:

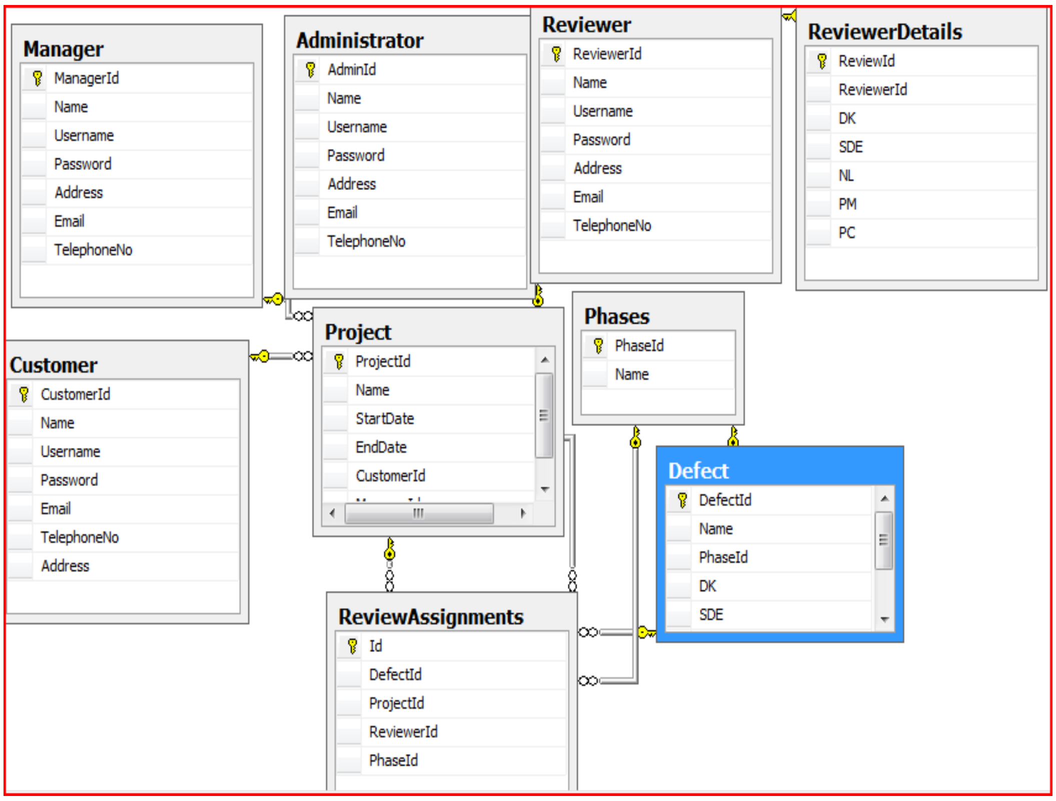


Fig 5.6: The database diagram

5.6 CLASS DIAGRAM

5.6.1 FTRLIBRARY STRUCTURE

As a business component, this project will not contain any graphical user interface. It will be built as a Class Library. Three types of classes will be developed for the FTRLibrary.

- Business Objects
- DAOs (Data Access Objects)

- Facade Objects

5.6.2 BUSINESS OBJECTS

These are abstract representations of entities from the business domain. They represent concepts that are important to the business that the system is modelling. In this system these are abstractions of Formal Technical Review related concepts, such as project, reviewer, defects, phase etc. The business objects in this component will be:

- Project

  Represents a project
- Defect

  Represents a defect.
- ReviewAssignments

  Represents the assigning of a reviewer to a defect.
- User

  Represents the general user of the system
- Manager

  Represents a project manager
- Reviewer

  Represents a reviewer.
- Customer

Represents a customer.

- ReviewerDetails

  Represents the background and experience information of a reviewer.

5.6.3 DATA ACCESS OBJECTS

Data Access Objects provide abstract interfaces to data sources, providing a clear separation between the business and persistence logic. The aim is to come up with a system that is highly robust and also has low coupling between our business classes and the database. The DAOs will contain all the SQL code for reading and writing to the database. There will be multiple DAOs, one DAO for each different user of the system.

- SuperDAO (super class for all others)
- AdminDAO
- ManagerDAO
- ReviewerDAO
- CustomerDAO

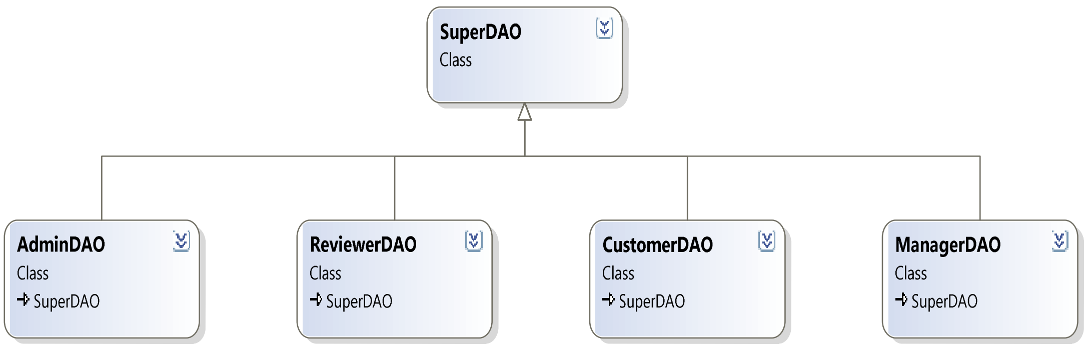


Fig 5.7: The Class Diagram for DAO

5.6.4 FACADE OBJECTS

The Facade classes will be used to provide a publicly available interface to the business component. One facade class for each type of user who will access the business component. This helps in that a user only sees what he/she needs to see. This enhances security of a distributed system. The facade classes are:

- FTRSuperFacade
- FTRAdminFacade
- FTRManagerFacade
- FTRReviewerFacade
- FTRCustomerFacade

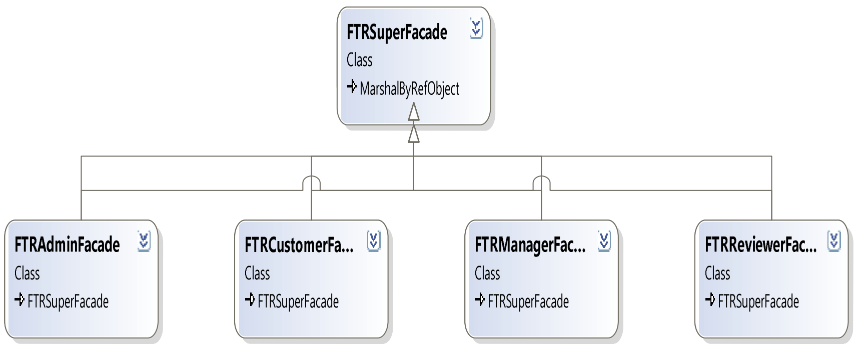


Fig 5.8: The Class Diagram for Façade Class

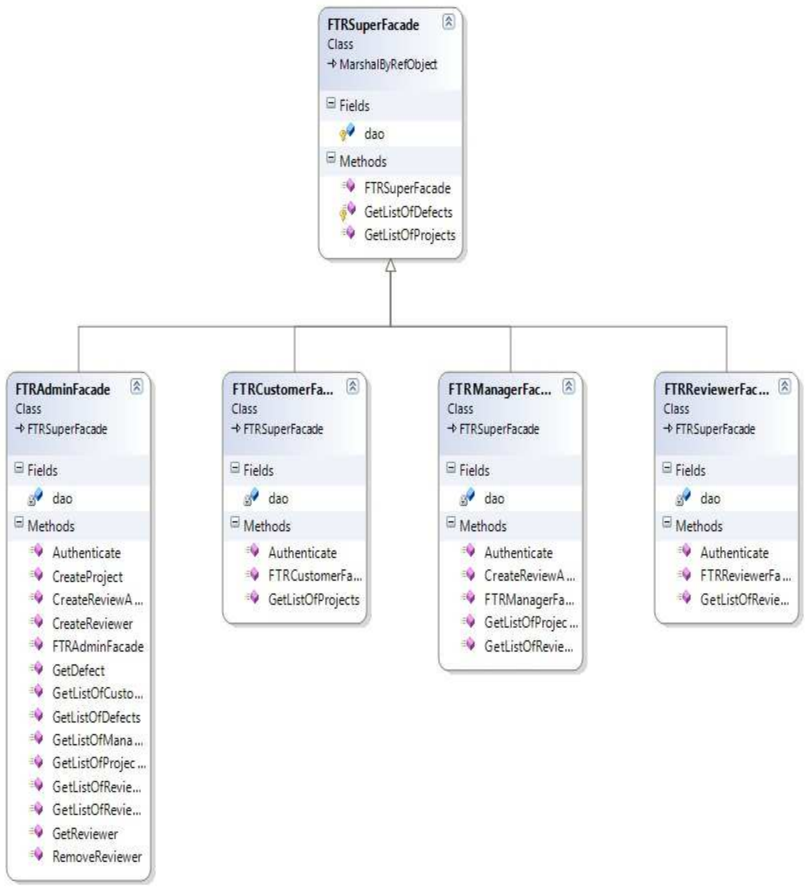


Fig 5.9: The Detailed Class Diagram for Façade Class

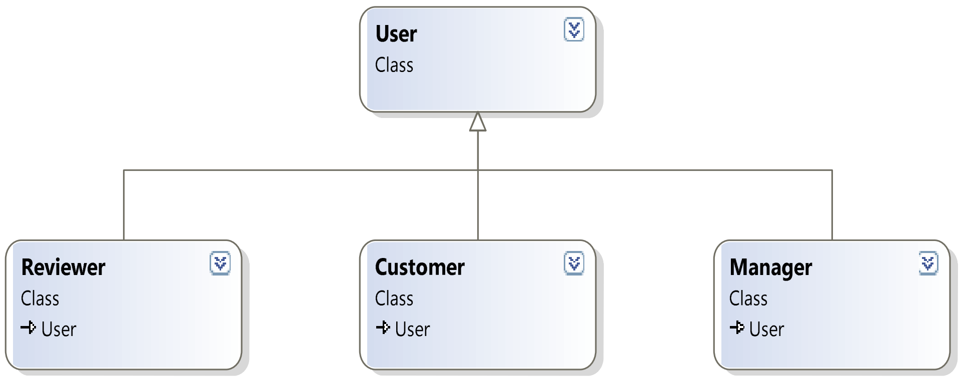


Fig 5.10: The Class Diagram for Users of the system

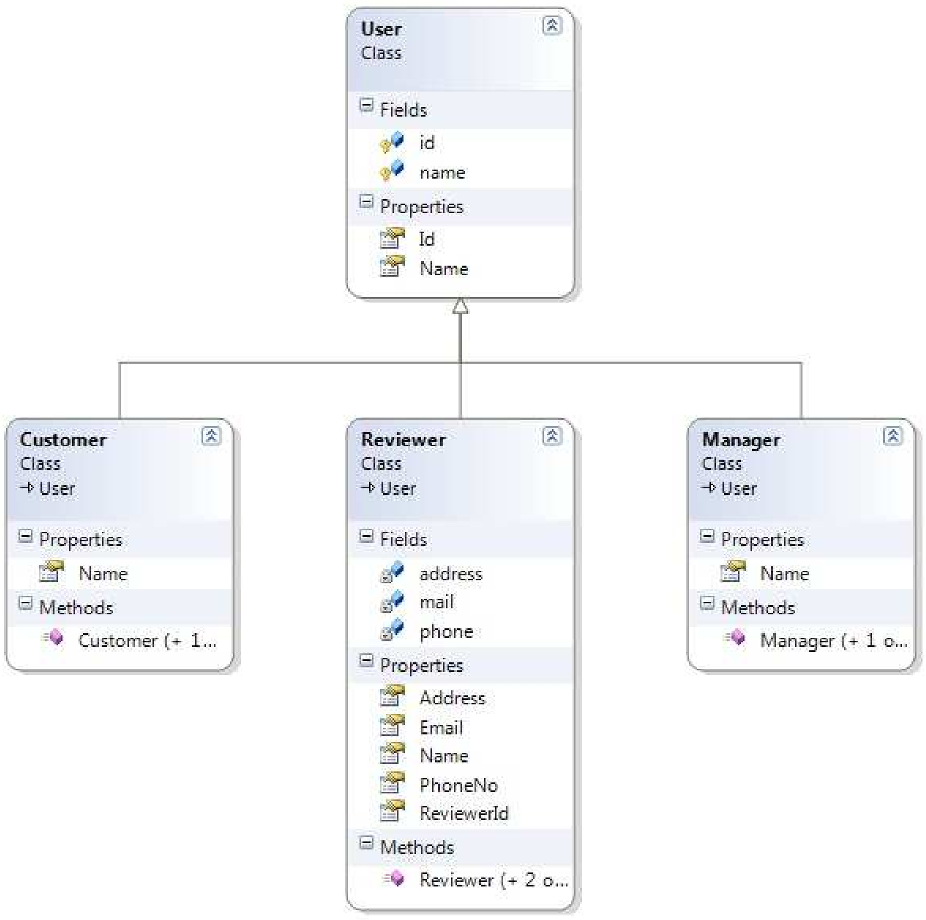
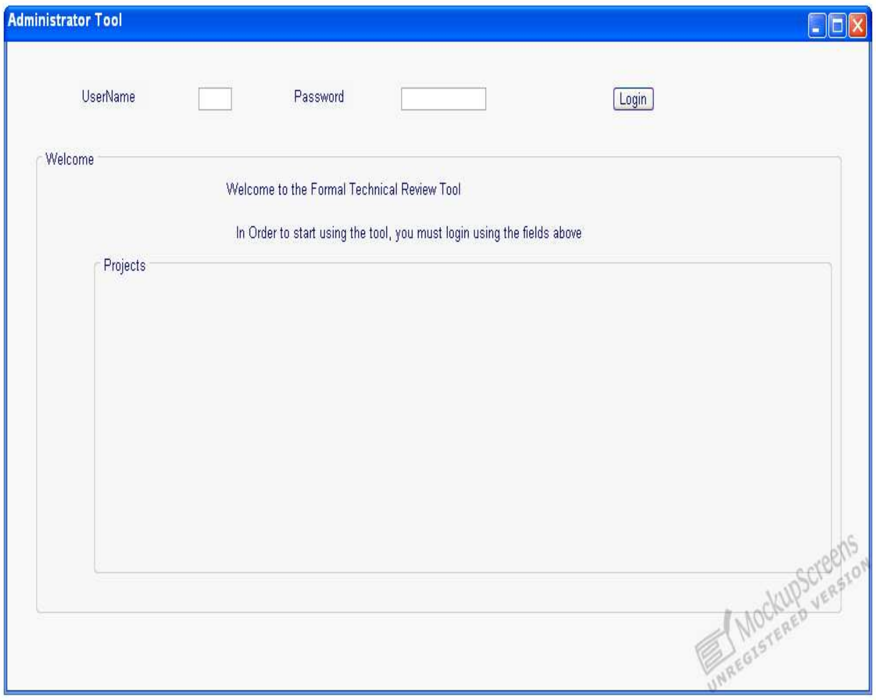


Fig 5.11: Detailed Class Diagram for Users of the system

ii) User interface

The mock up® screen tool was used to design the user interface for the system,



Welcome to the Formal Technical Review Tool

In Order to start using the tool, you must login using the fields above

Fig 5.12 User Interface for the administrator

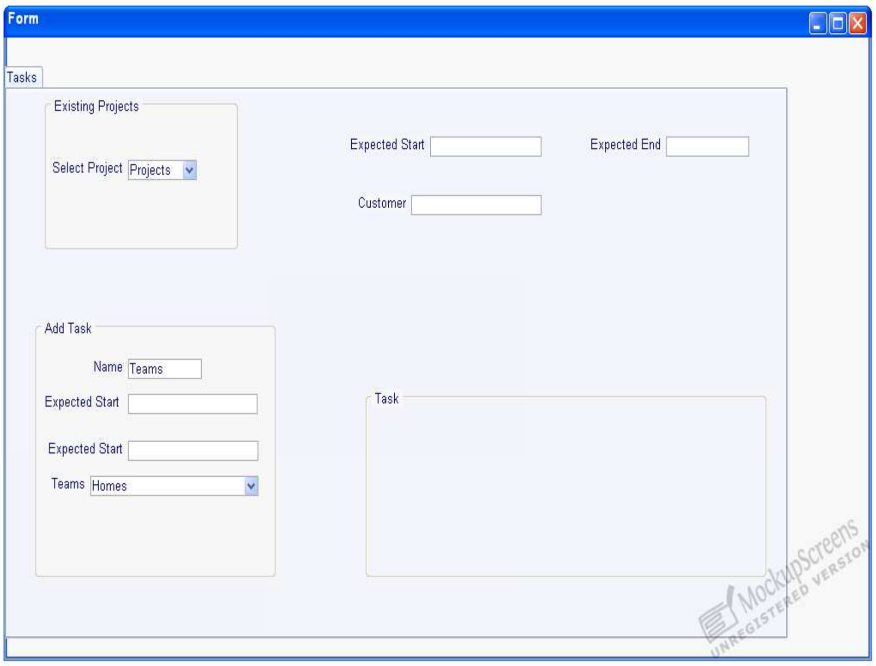


Fig 5.13 User Interface for the task Form

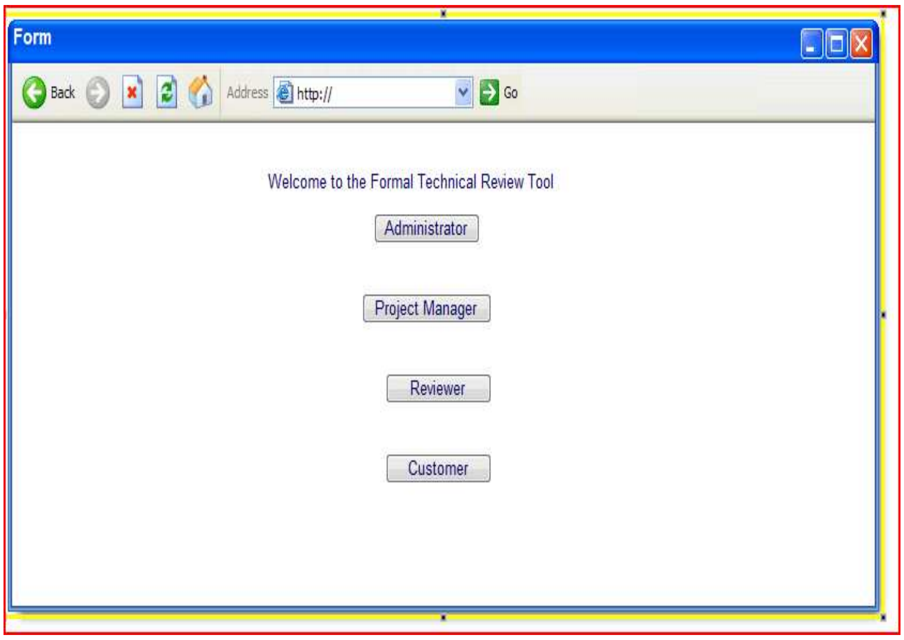


Fig 5.14 User Interface for the online application

The business component built and the GUI implemented have to communicate. This communication will be achieved using the .NET remoting. A remote server application will be created and will be used to distribute the admin tool and the FTRLibrary component. There are two ways of building distributed systems in .NET, which provide means to invoke an object on another computer via a local proxy [G.Coulouris05].

Web Services- works across platforms, so can be used to provide services to clients that are not under your control and could be written in any language.

.NET Remoting- works only when client and server are written in .NET, this can be used when both client and server are under your control.

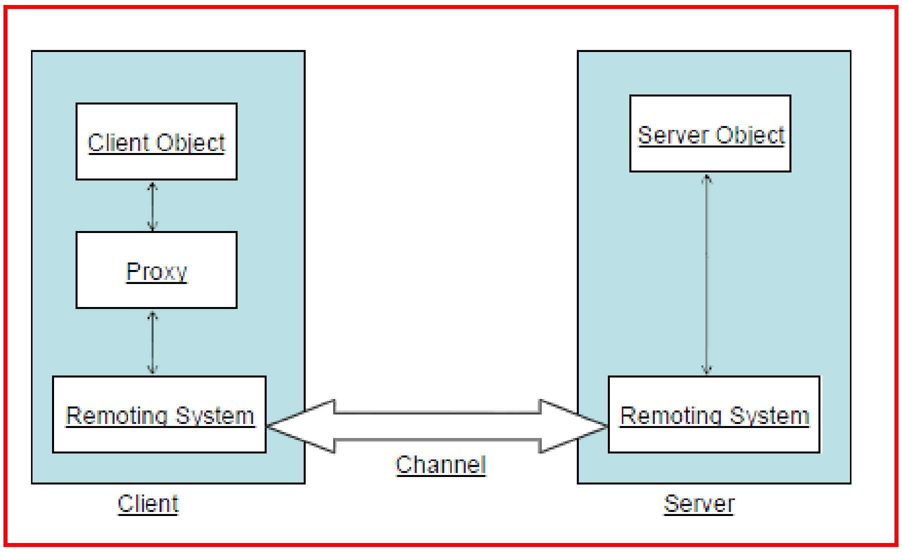


Fig 5.15: .NET Remoting Architecture

Marshalling- Marshalling determines how an object is exposed to the client application

Objects can be marshalled

- By value: a copy of the server object is sent and kept in the client domain
- By reference: the client only holds a reference to the object [G.Coulouris05]

In this application marshalling of the business objects will be by value. The objects will then reside on the client and calls to them will be faster than marshalling by reference.

“To do this, [Serializable] attribute is added to the class that is to be marshalled” [G.Coulouris05]

The superDao Code extracts are show below:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.Data.SqlClient;

namespace FTRLibrary.DAO
{
    public class SuperDAO
    {
       public List<Project> GetListOfProjects()
        {
            string sql;
            SqlConnection cn,cn2,cn3;
            SqlCommand cmd,cmd2,cmd3;
            SqlDataReader dr,dr2,dr3;
            List<Project> projects;
            projects = new List<Project>();

            sql = "SELECT * FROM Project";
            cn = new
SqlConnection(Properties.Settings.Default.FTRConnectionStrin
g);
            cmd = new SqlCommand(sql, cn);

            try{
                cn.Open();
                dr=cmd.ExecuteReader();
                while(dr.Read()){
                    sql="SELECT * FROM Customer WHERE
CustomerId="+(int)dr["CustomerId"];
                    cn2= new
SqlConnection(Properties.Settings.Default.FTRConnectionStrin
g);
                    cmd2 = new SqlCommand(sql, cn2);
                    cn2.Open();
                    dr2 =
cmd2.ExecuteReader(CommandBehavior.SingleRow);
                    dr2.Read();
                    Customer cust = new
Customer(dr2["Name"].ToString(), (int)dr2["CustomerId"]);
```

```
                    dr2.Close();

                    //
                    sql = "SELECT * FROM Manager WHERE
ManagerId=" + (int)dr["ManagerId"];
                    cn3 = new
SqlConnection(Properties.Settings.Default.FTRConnectionStrin
g);
                    cmd3 = new SqlCommand(sql, cn3);
                    cn3.Open();
                    dr3 =
cmd3.ExecuteReader(CommandBehavior.SingleRow);
                    dr3.Read();
                    Manager m = new
Manager(dr3["Name"].ToString(), (int)dr3["ManagerId"]);

                    dr3.Close();



                    Project p = new
Project((Guid)dr["ProjectId"], dr["Name"].ToString(), m,
cust,(DateTime)dr["StartDate"],(DateTime)dr["EndDate"]);

                    projects.Add(p);
```

```
                }
                dr.Close();
            }
            catch (SqlException ex)
            {
                throw new Exception("Error getting tasks
list", ex);
            }
            finally
            {
                cn.Close();
            }
            return projects;
        }
        public List<ReviewAssignments> GetListOfDefects(Guid
projectId)
        {
            string sql;
            SqlConnection cn, cn2;
            SqlCommand cmd, cmd2;
            SqlDataReader dr, dr2;
            List<ReviewAssignments> defects;
            defects = new List<ReviewAssignments>();

            sql = "SELECT * FROM DefectDetails WHERE
ProjectId='"+projectId+"'";
```

```
            cn = new
SqlConnection(Properties.Settings.Default.FTRConnectionStrin
g);
            cmd = new SqlCommand(sql, cn);

            try
            {
                cn.Open();
                dr = cmd.ExecuteReader();
                while (dr.Read())
                {
                    sql = "SELECT * FROM Defect WHERE
DefectId='" + dr["DefectId"]+"'";
                    cn2 = new
SqlConnection(Properties.Settings.Default.FTRConnectionStrin
g);
                    cmd2 = new SqlCommand(sql, cn2);
                    cn2.Open();
                    dr2 =
cmd2.ExecuteReader(CommandBehavior.SingleRow);
                    dr2.Read();
                    Defect d = new
Defect(dr2["DefectId"].ToString(),dr2["Name"].ToString());

                    dr2.Close();
                    ReviewAssignments dd = new
ReviewAssignments(dr["DefectId"].ToString(),d,(Phase)dr["Pha
seId"]);

                    defects.Add(dd);


                }
                dr.Close();
            }
            catch (SqlException ex)
            {
                throw new Exception("Error getting tasks
list", ex);
            }
            finally
            {
                cn.Close();
            }
            return defects;
        }
        }
```

The remote server code extract:

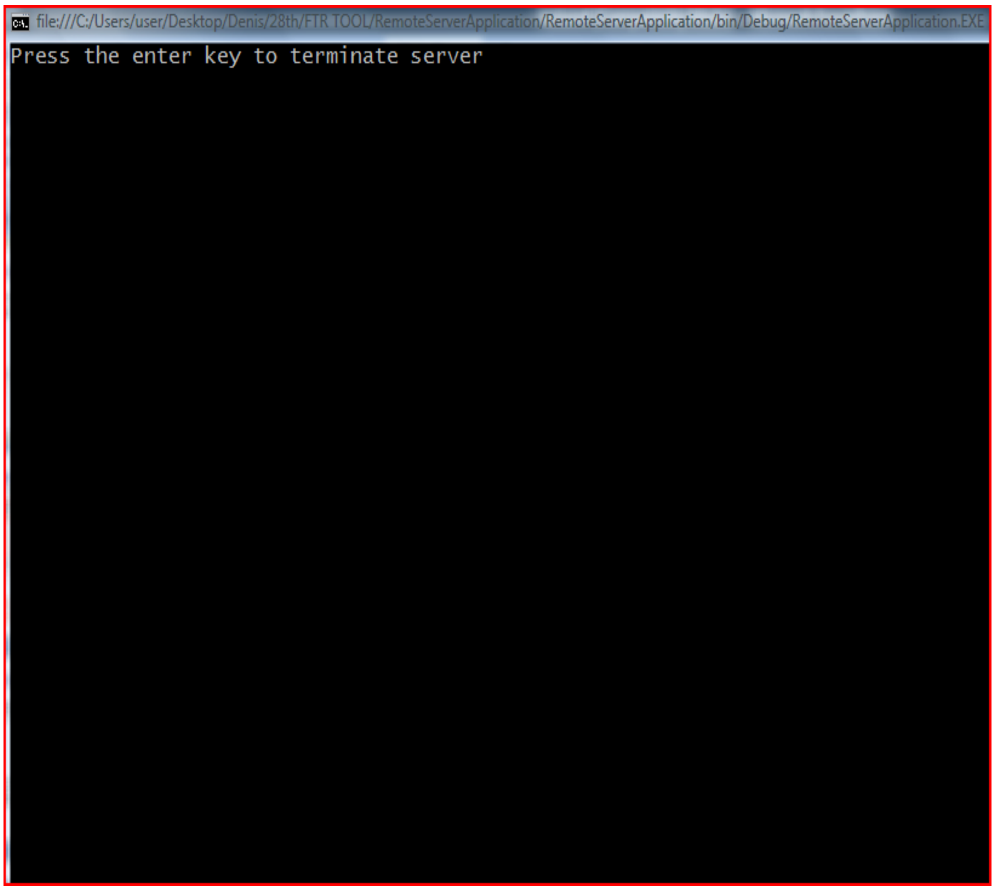
```
namespace RemoteServerApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            HttpChannel channel = new HttpChannel(50000);
            ChannelServices.RegisterChannel(channel, false);

RemotingConfiguration.RegisterWellKnownServiceType(typeof(FT
RLibrary.FTRAdminFacade), "FTRAdminFacade",
WellKnownObjectMode.Singleton);
            Console.WriteLine("Press the enter key to
terminate server");
            Console.ReadLine();



        }
    }
}
```

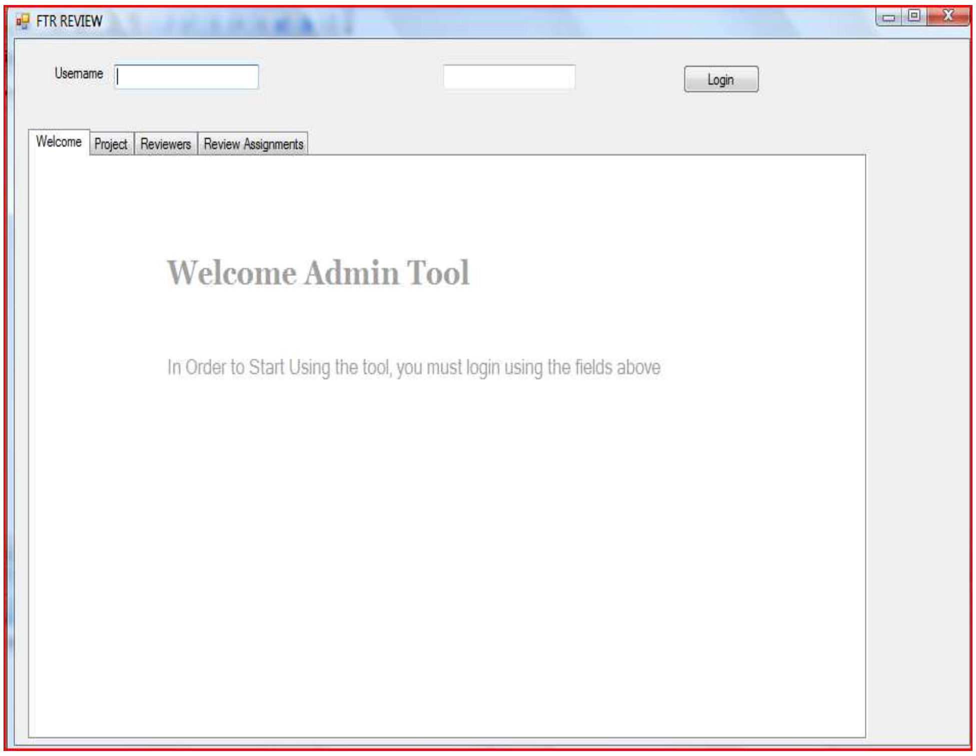
When the remote server application runs the following is the output



```
file:///C:/Users/user/Desktop/Denis/28th/FTR TOOL/RemoteServerApplication/RemoteServerApplication/bin/Debug/RemoteServerApplication.EXE
Press the enter key to terminate server
```

Fig 5.16: Output from the remote server application

After making sure that the remote server is running, the admin tool is started. The following is the output


FTR REVIEW

Username

Login

Welcome | Project | Reviewers | Review Assignments

Welcome Admin Tool

In Order to Start Using the tool, you must login using the fields above

Fig 5.17: First Page for login

After entering the correct password and username the tool will allow the admin to carry a number of functions as shown.

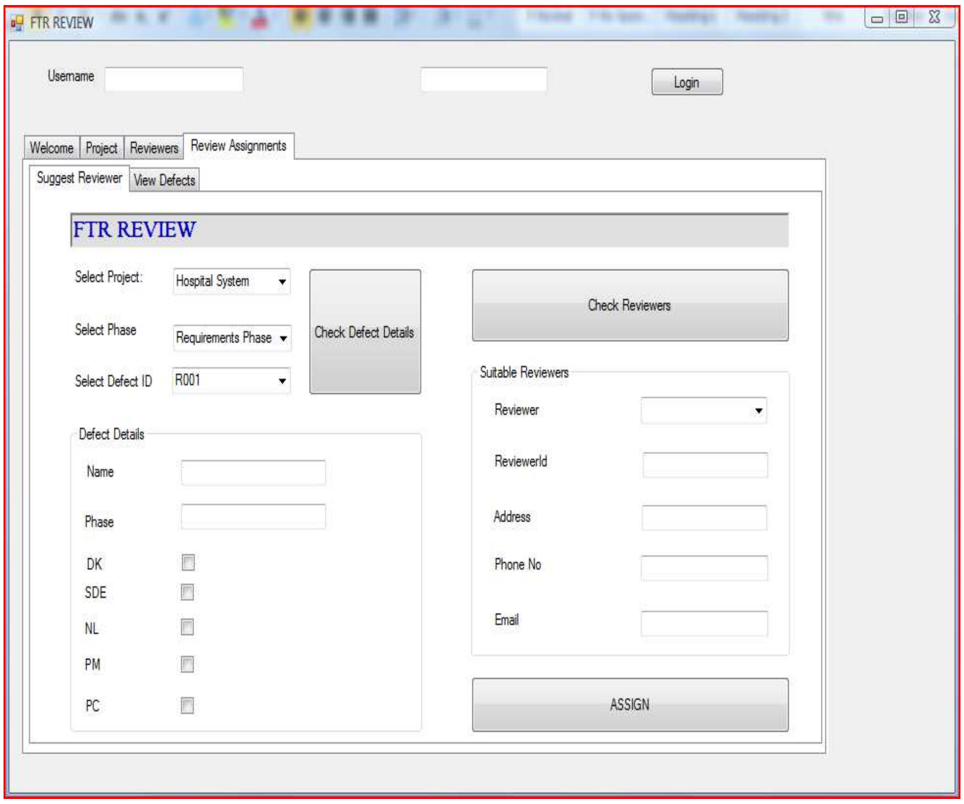


Fig 5.18: Output from the remote server application

5.7 Web services

Web services are self-contained, self-describing, modular applications that can be published, located, and invoked across the Web they can perform functions from simple requests to complicated business processes. Once a Web service is deployed, other applications (and other Web services) can discover and invoke the deployed service [IBM web service tutorial]

Web services are distributed computing model based on asynchronous messaging (XML), and they:

- Support dynamic application integration over the Web
- Web Services connect computers and devices with each other using the Internet to exchange data and access services
- On-the-fly software creation through the use of loosely coupled, reusable software components
- Business services can be distributed over the Internet

5.7.1 Web Service Architecture

i) Components

a) Service providers

- publish available services and offer bindings for services

b) Service brokers

- allow service providers to publish their services
- provide mechanisms to locate services and their providers

c) Service requestor

- uses the service broker to find a service and then
- invokes (or binds) the service offered by a service provider

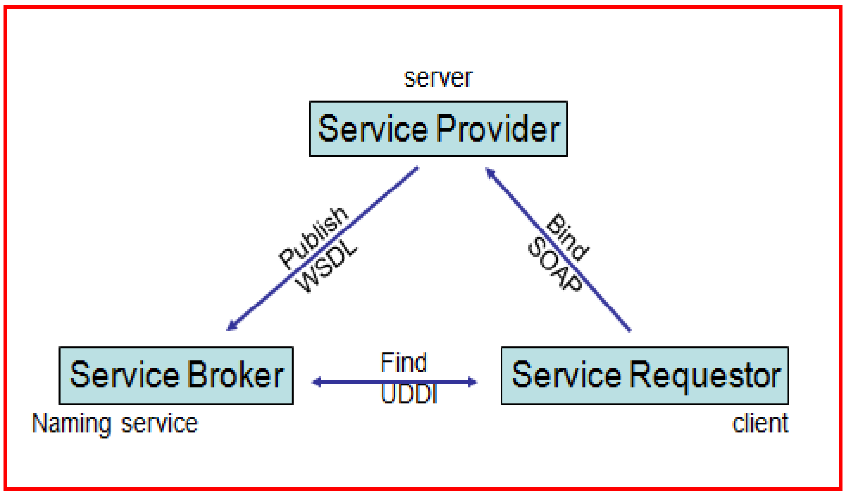


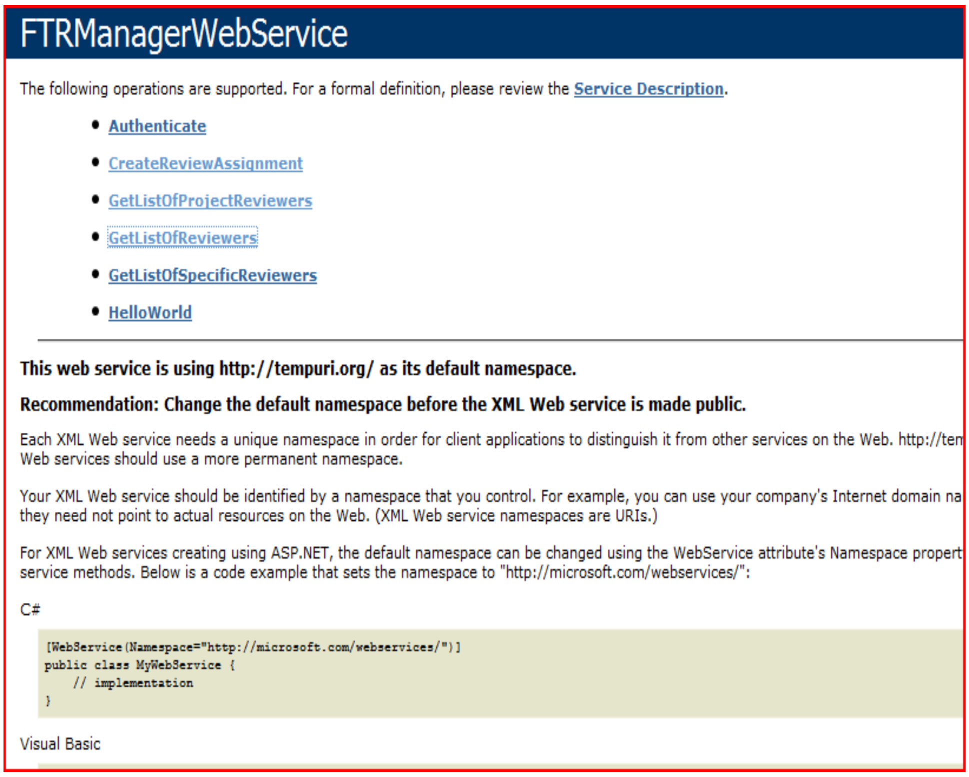
Fig 5.19: Illustration of Components in the web service

ii) Visual Studio Development Server

Visual Studio .NET 2008 comes with an in-built ASP.NET development server, this is the tool that will be used for development as it is possible to test your web services and websites without having to deploy to a web server

- The web server starts automatically when you run a web service or website project
- The address is http://localhost:*port* - Where port is a port number chosen by Visual Studio

Below is the diagram show the web Service functions of the admin:



FTRManagerWebService

The following operations are supported. For a formal definition, please review the Service Description.

- Authenticate
- CreateReviewAssignment
- GetListOfProjectReviewers
- GetListOfReviewers
- GetListOfSpecificReviewers
- HelloWorld

This web service is using http://tempuri.org/ as its default namespace.

Recommendation: Change the default namespace before the XML Web service is made public.

Each XML Web service needs a unique namespace in order for client applications to distinguish it from other services on the Web. http://ten
Web services should use a more permanent namespace.

Your XML Web service should be identified by a namespace that you control. For example, you can use your company's Internet domain na
they need not point to actual resources on the Web. (XML Web service namespaces are URIs.)

For XML Web services creating using ASP.NET, the default namespace can be changed using the WebService attribute's Namespace propert
service methods. Below is a code example that sets the namespace to "http://microsoft.com/webservices/":

C#

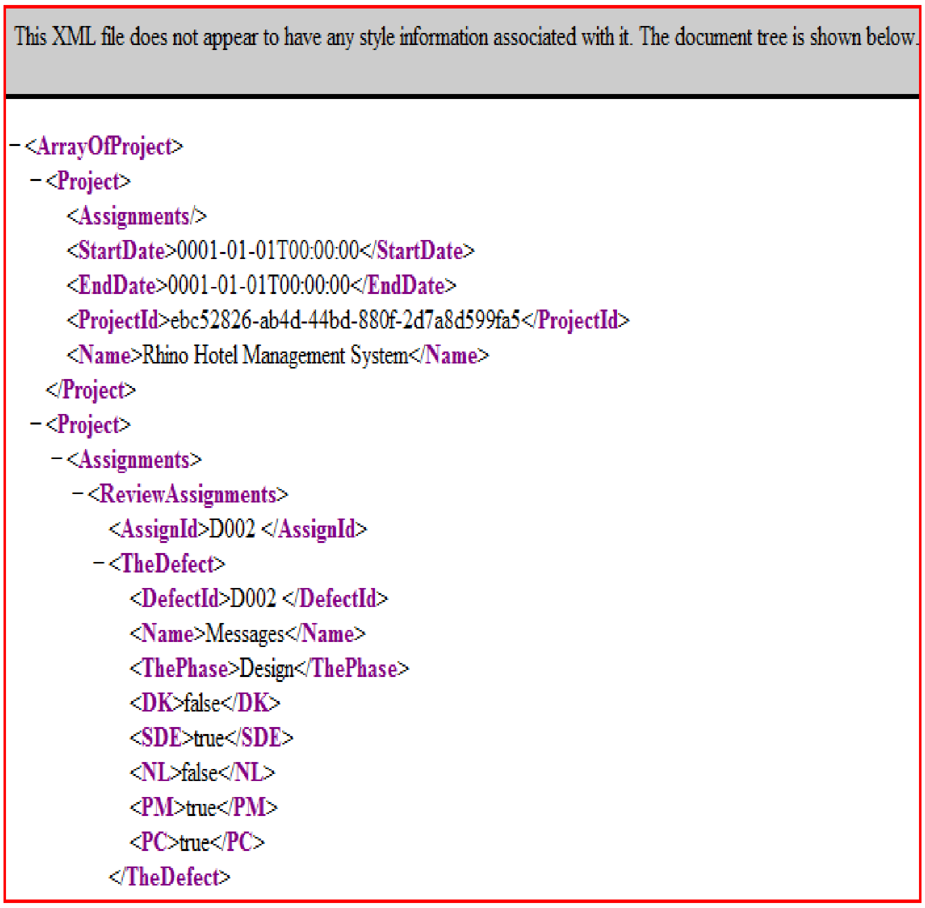
```
[WebService(Namespace="http://microsoft.com/webservices/")]
public class MyWebService {
    // implementation
}
```

Visual Basic

Fig 5.20: Methods exposed though a web service

When a web method is invoked, the message is returned in XML

(SOAP) e.g.



```
This XML file does not appear to have any style information associated with it. The document tree is shown below.

- <ArrayOfProject>
  - <Project>
      <Assignments/>
      <StartDate>0001-01-01T00:00:00</StartDate>
      <EndDate>0001-01-01T00:00:00</EndDate>
      <ProjectId>ebc52826-ab4d-44bd-880f-2d7a8d599fa5</ProjectId>
      <Name>Rhino Hotel Management System</Name>
    </Project>
  - <Project>
    - <Assignments>
      - <ReviewAssignments>
          <AssignId>D002 </AssignId>
        - <TheDefect>
            <DefectId>D002 </DefectId>
            <Name>Messages</Name>
            <ThePhase>Design</ThePhase>
            <DK>false</DK>
            <SDE>true</SDE>
            <NL>false</NL>
            <PM>true</PM>
            <PC>true</PC>
          </TheDefect>
```

Fig 5.21: XML message returned for the Projects

iii) Building the Customer Website

The last thing is to build a website that will allow admin, Manager, Reviewer and customers to log in and carry out some functions. The website will be built as an *ASP.NET Website* project (still in the FTRLibrary solution).

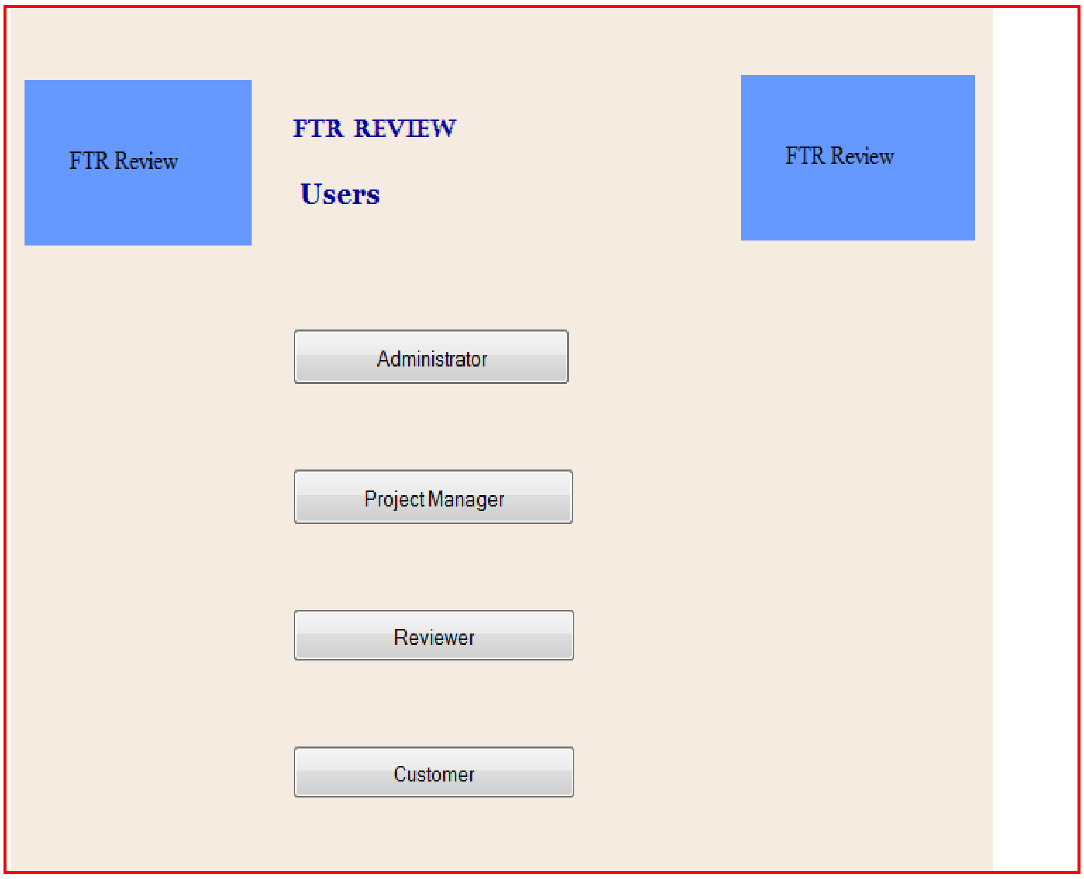
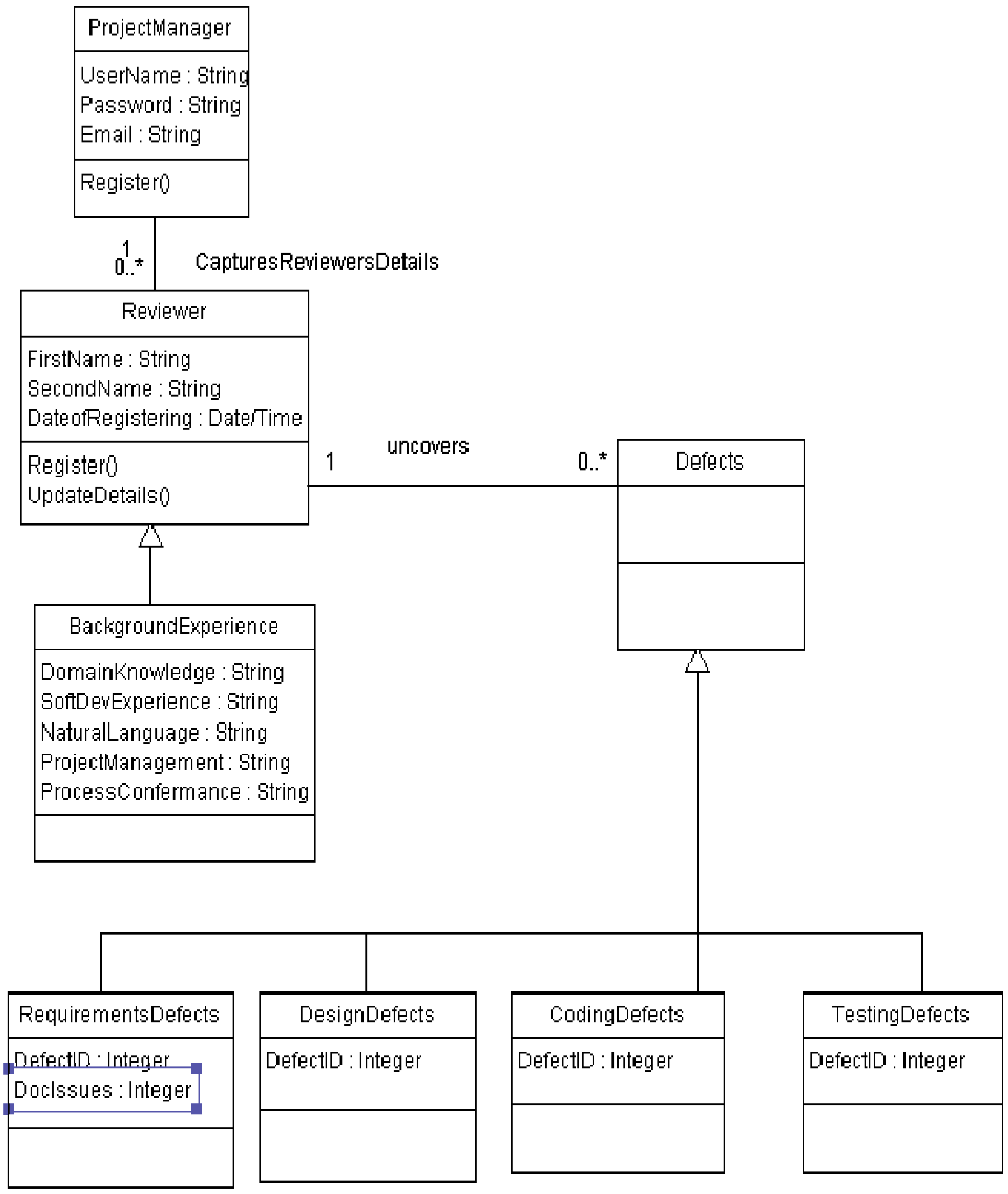


Fig 5.22: Web site Home page

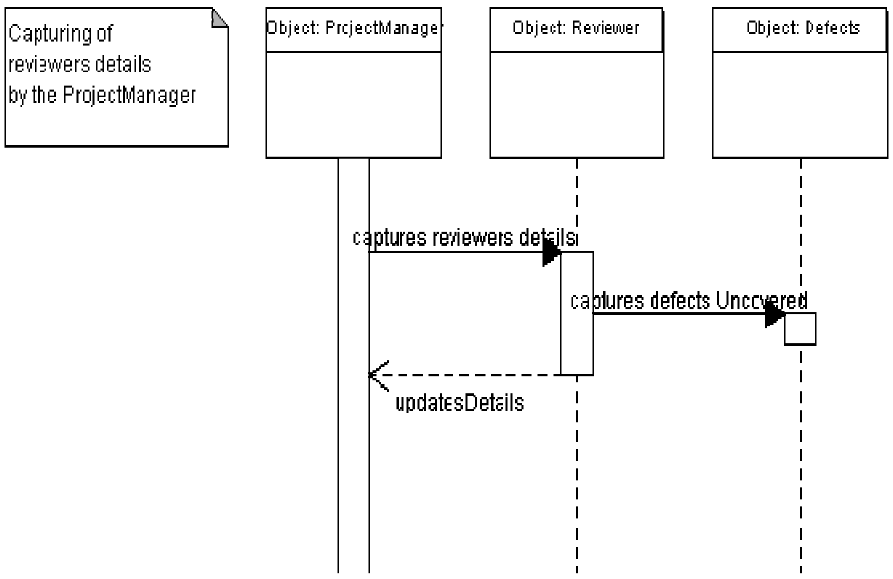
USER INTERACTION (PROJECT MANAGER)	**SYSTEM RESPONSIBILITY**
Register	System adds login details of the project manager as a registered user.
login	System logins in the registered user
Inputs the personal details of a reviewer	System stores those details in a database
Inputs the kinds of defects a certain reviewer has been able to uncover	This details are stored in a database
Retrieves information when organizing for a review	Details of which reviewer should help him/her uncover this defects are shown
Updates	Deletes or adds new information to the database
Logins off	Allows the user to log off.

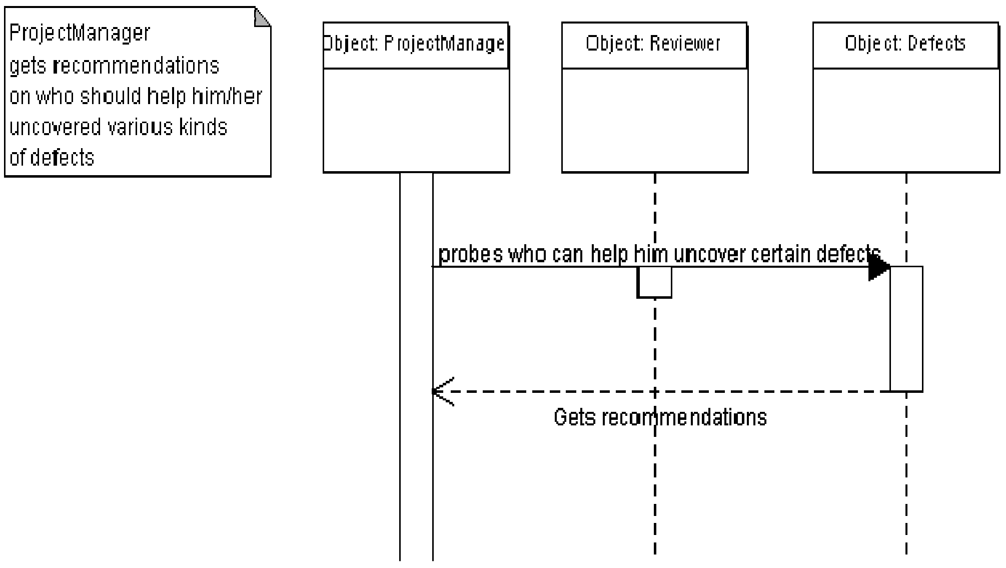
Table 5.1: Testing Results

Class Diagram

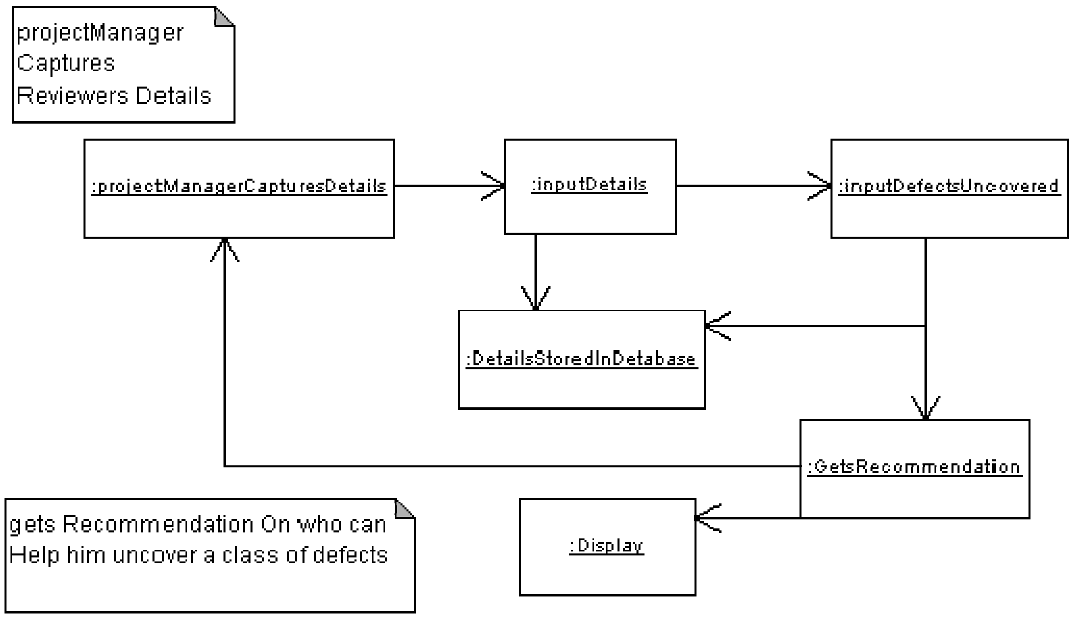


Sequence Diagrams:

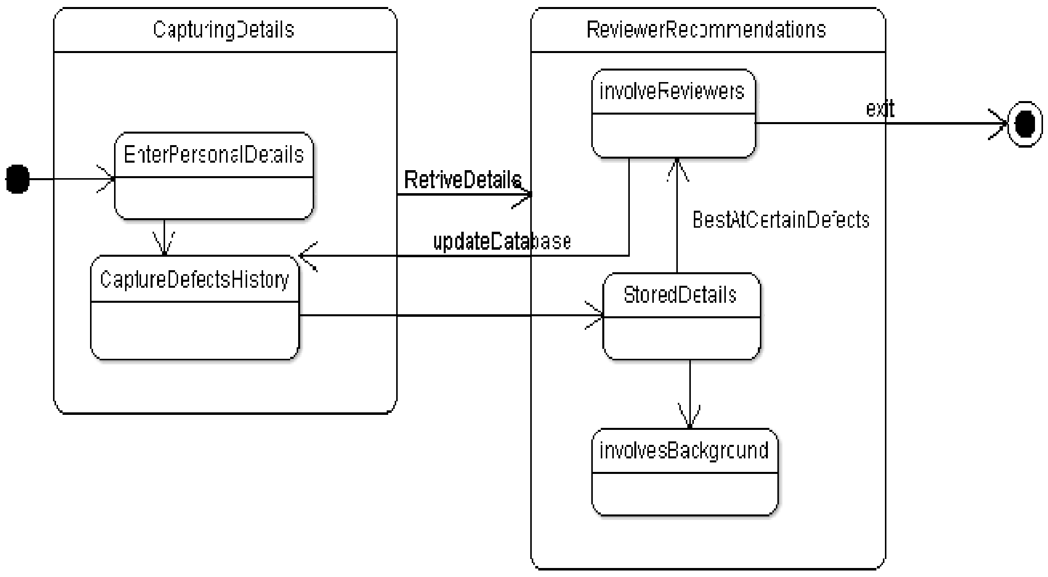



ProjectManager gets recommendations on who should help him/her uncovered various kinds of defects
Object: ProjectManage
Object: Reviewer
Object: Defects
probes who can help him uncover certain defects
Gets recommendations

Collaboration diagram:


projectManager Captures Reviewers Details
:projectManagerCapturesDetails
:inputDetails
:inputDefectsUncovered
:DetailsStoredInDetabase
:GetsRecommendation
:Display
gets Recommendation On who can Help him uncover a class of defects

Statechart Diagrams



5.8 TESTING

The following table shows test conditions and results for testing:-

S/N	**TEST**	**EXPECTED RESULT**	**ACHIEVED RESULT**
1	Web server User authentication	Web site to prompt user name and password	Prompted the user name and accepted the registered user
2	User authentication	Web site to give error message for non registered users	An error message was give on non registered users trying to login
3	Display results for the Hospitality Software Company (HSC)	Four web pages to display the case results of the HSC which can be edited	The results for the HSC development cycles displayed.
4	Records storage	Storage of personal and defects uncovered details in a database.	Capturing of details achieved
5	Giving Recommendations	Able to give a reviewer recommendations on who best can help him/her uncovered defects	User is able to get recommendations on who can help him uncover certain class of defects.

6	Editing of the stored data	The system should allow a user to modify the initial background and experience variables as well as the defects classes	The system allows modification of the database by the registered user.
6	Logs off	The system should allow the save details and exit	PASS

CHAPTER SIX

6.1 RECOMMENDATIONS AND CONCLUSION

Software products are built using a software process, and errors get introduced during the process for various reasons. The errors that are not detected and fixed during the software process reach customers and are called ‘defects’. Defects are often costlier to fix than errors and also damage the developer’s reputation. Therefore, it is necessary to reduce the errors in the end products. In order to deliver products of good quality, we need ‘quality filters’ that can be used to detect errors.

There are many review types under the broad category of FTRs. These vary in the degree of formality – walkthroughs and inspections being the most formal and casual meetings being the least formal. The more formal the review type, the more effective it is in finding errors. While various review types differ in their ‘exact’ activities and roles defined, there are some common steps and roles in a generic FTR. The common steps are:

Step 1: Pre–review meeting

Step 2: Review meeting

Step 3: Post review meeting

One of the main function in an FTR process is staffing. The generic roles are:

Moderator - leads the inspection, schedules meetings, controls the meetings, reports inspection results, and follows up on rework issues. Moderators should be trained in how to conduct inspections, including how to keep participants with strong technical skills but low social skills from killing each other.

Author - created or maintains the work product being inspected. The author may answer questions asked about the product during the inspection, and he also looks for defects. The author cannot serve as moderator, reader, or recorder.

Reader - describes the sections of the work product to the team as they proceed through the inspection. The reader may paraphrase what is happening in the product, such as describing what a section of code is supposed to do, but he does not usually read the product verbatim.

Recorder - classifies and records defects and issues raised during the inspection. The moderator might perform this role in a small inspection team. With the creation of the Formal Technical Review Support Tool, the job of the recorder is highly improved.

Inspector - attempts to find errors in the product. All participants actually are acting as inspectors, in addition to any other responsibilities.

This research had two main objectives, the first of which was to show that the reviewers experience and background affects the kind of defects reviewers are able to uncover. A typical FTR can benefit from the following activities:

- Planning for project reviews
- Training reviewers on how to participate in reviews
- Ensuring that the review meeting is effective
- Using review data to see how the review process can be further refined

It is a known fact that the above activities can add value to a Review meeting, (Ackerman89) there are some essential parameters for the meeting such as there should be an acceptable (between 3-5) number of persons conducting the meeting and that too after each one of them has done his/her homework i.e. some preparation and the meeting should not be carried out very long which may lead to wastage of time but rather for duration just enough to churn out some constructive results. FTR (Formal Technical Review) is effective when a small and specific part of the overall software is under scrutiny. It is easier and more productive to review in small parts like each module one by one rather than to review the whole thing in one go. The target of the FTR (Formal Technical Review) is on a component of the project, a single module.

The individual or the team that has developed that specific module or product intimates the product is complete and a review may take place. Then the project leader forwards the request to the review leader who further informs the reviewers who undertake the task.

It is as this stage that this study has found the importance of background and experience of the reviewers very important. In chapter Four the researcher has provided important summery of the background and experience variables and the kind of defects that those individual have a potentiality to uncover in HSC company. This is not expected to be uniform for all kinds of software products; this is so as software products are very different in a number of ways.

As a company engages in FTR over a period of time important data is generated which could provide set of heuristics or guideline with which the inspection manager can choose inspectors for his or her team. Based on the organization’s historical profile of defect types, he can suggest the background and experience that inspectors should have that will give them the best chance of finding those important defects. This then creates a very good case for keeping records.
The second objective of this study was to develop a tool for helping in keeping records.

6.2 KEEPING RECORDS

Record keeping is a major distinction between informal and formal review activities. There are three aspects to this task: recording defects during the inspection meeting; collecting data from multiple inspections; and analyzing the defect trends to assess inspection effectiveness and identify ways of improving software development process to prevent common types of defects.

Many books contain sample inspection recording forms; the FTR support tool can help the project manager in the keeping of records. As inspectors raise issues during the review meeting, the recorder enters them on the issues list from The management report contains information about the material that was inspected and the disposition of the product (accepted with minor changes, etc.), but no actual information about the defects found is included. The purpose is to allow managers to know how the project is progressing and to look for areas where improvements should be made. The moderator usually is responsible for preparing these post-inspection reports. With the Formal Technical Review Support Tool one is able to produce reports after inspections have been carried out.

An effective, ongoing inspection process permits an organization to combine data from multiple inspections to gain insight into the quality of both the review process and the products being reviewed. The ultimate objective is to have a database of inspection data so that quantitative conclusions can be drawn from defect trends and inspection process information.

While results will vary from one organization to the next for many reasons, if you begin recording and analyzing your inspection data, you will be able to determine which methods work best for you and you can begin to assess the quality of your work products.

Implementing software inspections is an important step along the path to a more mature software development process. In the cycle of continual process improvement that leads to concurrent improvements in both quality and productivity, inspections can play a major role.

In conclusion the research proposes the following framework for carrying out Formal Technical Review:

1. The producer indicated they are ready for a review
2. Having knowledge about the most important defects that the software development house wants uncovered then the manager selects reviewers with experience and background that can help the organization uncover those important defects.
3. The reviewer(s) receive the artefacts to be reviewed
4. The reviewers spend 2 hours inspecting the artefacts
5. The review takes place
   a) The producer walks through their product
   b) The reviewer(s) ask questions on the walk through
   c) The reviewers ask questions from their notes
   d) This details are captured in the Formal Technical Review Support Tool (FTR), that helps to automate the recording keeping process.
   e) A set of recommendations is produced and given to the producer to enable improvement

6.3 CONCLUSION

There is little or no doubt that formal technical reviews are one of the best methods for detecting defects at the earliest possible stage of the software development life cycle. This makes software inspections a major part of software quality as a whole, as the ability to detect and remove defects is not only vital but also the earlier found the more cost and time efficient for the development company.

This research has therefore shown that the software review meeting can be improved by involving reviewers with background and experience that has a higher chance of uncovering a certain kind of defect that the software development company considers “important”. A software tool has also been developed to improve record keeping and give suggestions on the most important background and experience for a certain kind of defect.

REFERENCES

Ackerman, A., Buchwalk, L. and Lewski, F. (1989) 'Software Inspections: An Effective Verification Process.' *IEEE Software*, May.

Basili, V. and Weiss, D. (1981) 'Evaluation of a Software Requirements Document by Analysis of Change Data.' In Proceedings 5th Int'l Conference on Software Engineering, *IEEE CS Press*, Mar.

Bassin, A., Kratschmer, T. and Santhanam P. (1998) 'Evaluating Software Development objectively.*' IEEE Software*, vol. 15, no 6, pp. 66-74.

Birk, A. and Tauz, C. (1998) 'Knowledge Management of Software Engineering Lessons Learned.' Proceedings of the Tenth Conference on Software Engineering and Knowledge Engineering, *Illinois, Skokie: Knowledge Systems Institute*, pp. 24-31.

Brad, C. and Dave, Z. (2001) *How Good is the Software: A review of Defect prediction Techniques*, Carnegie Mellon University, pp 9-10.

Brennan, R. and Prediger, D. (1981) *Coefficient λ: Some Uses, Misuses, and Alternatives*, Educational and Psychological Measurement, pg 41, 687-699.

Brocklehurst and Littlewood (1992) ‘New Ways to Get Accurate Software Reliability Modeling.*’ IEEE Software*, vol. 34, no. 42.

Bush, M. (1990) ‘Improving Software Quality: The Use of Formal Inspections at the Jet Propulsion Laboratory.’ Proc. 12th Int’l Conf. Software Eng., *IEEE CS Press*, Los Alamintos.

Byrt, T. Bishop J and Carlin JB (2009) ‘Bias, prevalence and kappa.’ Journal of Clinical Epidemiology 46: 423.

Chillarege R., Bhandari I.S., Chaar, J.K., Halliday M.J., Moebus D.S., Ray B.K. and Wong M. (1992) ‘Orthogonal Defect Classification: A Concept for In-Process Measurements.’ *IEEE Transactions on Software Engineering*, vol 18, no 11, pp. 943-956.

Cohen and Jacob, A. (1960) ‘A coefficient of agreement for nominal scales.’ *Educational and Psychological Measurement*’, Vol.20, No.1, pp.37-46.

Day I, (1993) ‘Qualitative data analysis.’ *A user-friendly guide for social scientists*, *New York*: Routledge.

Doolan, E.P. (1992) ‘Experience with Fagan’s Inspection Method.’ *Software—Practice and Experience*, vol. 22(2), February.

Eickelmann, Nancy S, Ruffolo, Francesca, Baik, Jongmoon and Anant (2003) ‘An Empirical Study of Modifying the Fagan Inspection Process and the Resulting Main Effects and Interaction Effects Among Defects Found, Effort Required, Rate of Preparation and Inspection, Number of Team Members and Product 1st Pass Quality.’ Proceedings of the 27th Annual NASA Goddard, *IEEE Software Engineering Workshop*.

F. Akiyama (1971) ‘An Example of Software System Debugging.’ *Information Processing*, vol. 71, pp. 353-379.

Fagan, M. E. (2000) ‘Design and code inspections to reduce errors in program development.’ *IBM Systems Journal*.

Fagan, M. E. (1986) ‘Advances in Software Inspections.’ *IEEE Transactions on Software Engineering*.

Ferdinand, (1974) ‘A Theory of System Complexity.’ *Int’l J. General Systems*, vol. 1, pp. 19-33.

Fredericks. (2006). Detecting Defects in Object-Oriented Designs. *Using Reading Techniques to Increase Software Quality* (pp. 47-56). Denver: ACM Press.

Gilgun, J.F. (1992) *Definitions, Methodologies, and Methods in Qualitative Family Research Qualitative Methods in Family Research* pp. 22-29.

Glaser, B. G., and Strauss, A.L. (1967) The Discovery of Grounded Theory: Strategies for qualitative research, New York: Aldine de Gruyter.

Group, T. S. (2009). *Standish Group report* . Boston : CHOAS .

http://www1.standishgroup.com/newsroom/chaos_2009.php

http://databases.about.com Web portal containing links to a variety of database issues

http://www.nwlink.com/~Donclark/hrd/history/hawthorne.html [accessed on 3rd February 2009 at 10.00 am]

Johnson, P.M. (1996) "*Design For Instrumentation: High Quality Measurement Of Formal Technical Review*". Software Quality Journal volume 5, number 1.

K. Koga (1992) ‘Software Reliability Design Method in Hitachi.’ *Proc. Third European Conf. Software Quality*, Madrid.

K. Yasuda (1989) ‘Software Quality Assurance Activities in Japan.’ *Japanese Perspectives in Software Eng.*, pp. 187-205, Addison-Wesley.

Kathy Charmaz*, (2006)* ‘Constructing Grounded Theory: A Practical Guide through Qualitative Analysis.’ *Pine Forge Press*.

M. Dyer (1992) *The Clean room Approach to Quality Software Development.* Wiley.

Mitchell Scott (2007), Examining ASP.NET 2.0’s Membership, Roles and Profiles Part 8. Available at: http://aspnet.4guysfromrolla.com/articles/010307-1.aspx [Accessed 08 March 2010].

Mitchell Scott (2007), Examining ASP.NET 2.0’s Membership, Roles and Profiles Part 1. Available at: http://aspnet.4guysfromrolla.com/articles/120705-1.aspx [Accessed 01 January 2011].

MSDN Library (2009), ASP.NET Developer centre, Shared Code Folders in ASP.NET websites. Available at: http://msdn.microsoft.com/enus/library/t990ks23 (VS.80).aspx [Accessed 03 March 2010] http://www.mathworks.com/matlabcentral/fileexchange/15426 [Accessed 04 December 2009]

M. Halstead (1977) *Elements of Software Science.* New York Elsevier, North-Holland.

N.E. Fenton, S. Lawrence Pfleeger, and R. Glass (1994) 'Science and Substance: A Challenge to Software Engineers.' *IEEE Software*, pp. 86-95, July.

N.E. Schneidewind and H. Hoffmann (1979) 'An Experiment in Software Error Data Collection and Analysis.' *IEEE Trans. Software Eng.*, vol. 5, no. 3, May.

Nakajo and Kume (1991) 'A Case History Analysis of Software Error Cause-Effect Relationships.' *IEEE Trans. Software Eng.*, vol. 17, no. 8, Aug.

Pankaj Jalote (2002) *Software Project Management in Practice*, Addison-Wesley, pg 119.

Philip M. Johnson (1988) 'Supporting technology transfer of formal technical review

Potier, D., Albin, M., Ferreol, A. and Bilodeau (1982) 'Experiments with Computer Software Complexity and Reliability.' *Proc. Sixth Int'l Conf. Software Eng.*, pp. 94-103.

Pressman, R., 2005. *Software Engineering: A Practitioner's Approach*. Singapore: McGraw-Hill.

Royce, W.W., 1987. *Managing the development of large software systems: concepts and techniques*. Los Alamitos: IEEE Computer Society Press.

Sfetsos, P., Angelis, L. & Stamelos, I., 2006. Investigating The Extreme Programming System - An Empirical Study. *Empirical Software Engineering*, 11(2), pp.269-301.

R.D. Buck and J.H. Robbins (2008) 'Application of Software Inspection Methodology in Design and Code', *Software Validation, Northampton Square*.

R. W. Selby and A. A. Porter, Learning from examples: Generation and evolution of decision trees for software resource analysis.' *IEEE Trans. on Software Engineering*, vol. 14, December.

R.B. Grady (1992) '*Practical Software Metrics for Project Management and Process Improvement.'* Prentice Hall.

R.D. Buck and J.H. Robbins (2008) 'Application of Software Inspection Methodology in Design and Code', *Software Validation, Northampton Square*.

Schneider, G.M., Martin, J., and Tsai, W.T. (1992) ‘An Experimental Study of Fault Detection in User Requirements Documents.’ *ACM Transactions on Software Engineering and Methodology*, Vol. 1, April

Shneiderman, B. (1998) Designing the User Interface: *Strategies for Effective Human-Computer Interaction*: Addison-Wesley.

Sim, J. and Wright, C. C. (2005) ‘The Kappa Statistic in Reliability Studies.’ *Use Interpretation, and Sample Size Requirements in Physical Therapy*, Vol. 85, pp. 257-268.

Singer, J. and Lethbridge, T.C. (1996) ‘Methods for Studying Maintenance Activities.’ In Proceedings of the 1st International Workshop on Empirical Studies of Software Maintenance, *Monterey*, CA.

Steve Borgatti (1990) *Introduction to Grounded Theory*: Grounded Theory Institute.

T. J. McCabe (1976) ‘A complexity measure.’ *IEEE Trans. On Software Engineering*, vol. SE-2, no. 4.

Tauriainen (1999) *Experience Capturing Process and Its Enactment: Master Thesis*. University of Oulu, Department of Information Processing Science.

Tom Gilb (1993) ‘Software Inspection.’ *Addison-Wesley*.

University of Greenwich (2008) *Research Ethics Policy* available from http://www.gre.ac.uk/research/research_ethics_committee/policy [Accessed: 31st October 2010].

University of Greenwich (2008) *Study Skills – Referencing* available from http://www.gre.ac.uk/study_skills/referencing) [Accessed: 31st October 2010].

VanSomeren, M., Bernard, Y.F. and Sandberg (1994) ‘The Think Aloud Method: A Practical Guide to Modeling Cognitive Processes.’ *Academic Press*.

Veevers, A. (1994) ‘A Relationship between Software Coverage Metrics and Reliability.*’ Software Testing Verification and Reliability,* vol. 4, pp. 3-8.
Ward John & Peppard Joe (2002), Strategic Planning for Information Systems. 3rd edition. England: John Wiley & Sons

Wiegers, K. E. (2008). *Improving Quality Through Software Inspections.* Chicago: Adventure Works Press.

APPENDIX

AN INTRODUCTION LETTER

Lawrence Nderu

ICSIT, JKUAT

Dear Respondent,

I am a Post Graduate student (M.Sc. Software Engineering) at Jomo Kenyatta University of Agriculture and Technology (JKUAT), carrying out a research study on, "**FORMAL TECHNICAL REVIEW IN SOFTWARE QUALITY ASSURANCE**". The questionnaire (attached) aims at establishing the effects of background and experience of a reviewer on the defects uncovered during a ***Formal Technical Review (FTR).*** The information on the questionnaire will be kept ***Confidential*** and no individual names will be mentioned in the completion of the work. Your answers in this study will go along way in assisting the researcher understand the relationship between the background and experience of the reviewers and the defects uncovered during a FTR.

Thanking you in advance.

Yours faithfully

Lawrence Nderu

Researcher

Questionnaire

Name ____

Section A

General Background (tick the most appropriate)

1. My reading comprehension skills are:
    - o Low ☐
    - o Medium ☐
    - o High ☐
2. My listening and speaking skills are:
    - o Low ☐
    - o Medium ☐
    - o High ☐
3. What is your previous experience with software development in practice? (Check the bottom- most item that applies.)
    - o I have never developed software. ☐
    - o I have developed software on my own. ☐
    - o I have developed software as a part of a team, as part of a course. ☐
    - o I have developed software as a part of a team, in industry one time. ☐

- I have worked on multiple projects in industry. ☐

i) **Please explain your answer**. Include the number of semesters or number of years of relevant experience. (E.g. “I worked for 10 years as a programmer in industry”; “I worked on one large project in industry”; “I developed software as part of class project”; etc…)

____

Software Development Experience

Please rate your experience in this section with respect to the following 5-point scale:

(Please include any relevant comments below each section)

1 = No experience; **2** = learned in class or from book; **3** = used on a class project;

4 = used on one project in industry; **5** = used on multiple projects in industry

Experience with Lifecycle Models

4. Experience with the *Waterfall Model* 1 2 3 4 5

Comment: ____

5. Experience with the *Spiral Model* 1 2 3 4 5

Comment: ____

6. Experience with the *Incremental Development Model* 1 2 3 4

5

Comment: ____

Experience with Requirements

Experience writing requirements 1 2 3

4 5

Experience interacting with users to write requirements 1 2 3

4 5

Experience writing use cases 1 2 3

4 5

Experience reviewing requirements 1 2 3

4 5

Experience reviewing use cases 1 2 3

4 5

Experience changing requirements for maintenance 1 2 3

4 5

Comments: ____

Experience in Design

Experience in design of systems	1	2	3	4	5
Experience in design of systems from requirements/use cases	1	2	3	4	5
Experience with creating Object-Oriented (OO) designs	1	2	3	4	5
Experience with creating Structured Designs	1	2	3	4	5
Experience with reading OO designs	1	2	3	4	5
Experience with the Unified Modeling Language (UML)	1	2	3	4	5
Experience changing designs for maintenance	1	2	3	4	5

Comments: ____

Experience in Coding

Experience in coding, based on requirements/use cases	1	2	3	4	5
Experience in coding, based on design	1	2	3	4	5
Experience in coding, based on OO design	1	2	3	4	5
Experience in maintenance of code	1	2	3	4	5

Comments: ____

Experience in Testing

Experience in testing software	1 2 3 4 5
Experience in testing, based on requirements/use cases	1 2 3 4 5
Experience with Unit Testing	1 2 3 4 5
Experience with Integration Testing	1 2 3 4 5
Experience with System Testing	1 2 3 4 5

Other Experience

• Experience with software project management?	1 2 3 4 5
• Experience with software inspections?	1 2 3 4 5

Comment: ____

Section B

i) Please rate how much you think the background and experience (column) factor affects the ability of the reviewer to uncover the defects categories (rows) in this section with respect to the following 5-point scale: (Please include any relevant comments)

1 = Strongly Agree; 2 = Agree; **3** = Disagree; **4** = Strongly Disagree;

5 = Do not Know

<u>NOTATION</u>

DK – Amount of knowledge about the domain containing the problem being solved.

SDE - Experience in whatever development phase the inspection is occurring at**.**

NL - Familiarity with the documents language.

PM – Experience in software projects management

PC- How closely the inspector uses the checklist or the process provided.

a) Requirements Phase

Experience and background of the reviewer (Coded as above)						
		DK	**SDE**	**NL**	**PM**	**PC**
Defects classes under requirements phase	Document Formatting issues					
	Terminology – incorrect terminology					
	States- deals with system, object or data states					
	Data –inconsistence in data types, validation, access and restrictions					
	Interface and Access- way of accessing the system also interface					
	Other Inconsistencies – Contradictory things in two parts of the documents					

b) Design Phase

Experience and background of the reviewer (Coded as above)						
		DK	**SDE**	**NL**	**PM**	**PC**
Defects classes	**Messages** – problems involving messages that appear in the design.					
	Attributes – Description of attributes in the design.					

	Actors – Misuse of the actors within the design.					
	Constraints – Constraints are omitted or incorrect					
	Relationships –Incorrectly related classes.					
	Class Hierarchy –Problems within the class inheritance structure.					
	Other –Defects that do not fit in other categories.					
	States- states of the objects unclear					

c) Coding Phase

Experience and background of the reviewer (Coded as above)						
Defects classes under Coding phase		**DK**	**SDE**	**NL**	**PM**	**PC**
	Logic- insufficient/incorrect algorithms used.					
	Standards – problems with coding/documentation standards.					
	Redundant code					
	Performance – poor processing speed: System crash because of file size, Memory problems					
	Reusability – inability to reuse the code					
	Memory management defects – e.g. array overflow, illegal function call, system hands, or memory					

	overflow					
	Consistency – Updating or deleting of records in the same order throughout the system					
	Portability – code not independent of the platform					

.

d) Testing Phase

Experience and background of the reviewer (Coded as above)						
Defects classes under Testing phase		**DK**	**SDE**	**NL**	**PM**	**PC**
	Testing Tools and resources – Are the tools and resources identified and available.					
	Record keeping – has the record keeping mechanism being established.					
	Stress Testing Mechanisms – is the stress testing mechanism established?					
	Major identification of test phases – identification and sequencing of major test phases and the sequencing.					

SYSTEM HARDWARE AND SOFTWARE PACKAGES USED FOR THIS PROJECT

- Desktop Computer – 2.8 GHZ, 40GB, 512 MB
- Windows Xp Service pack 3
- Visual Studio 2008
- Microsoft SQL server 2008
- Mockup Screen development Tool
- AgroUML
- Apache Ant