Software Issue Tracking and Management System

For

Software Operational and Maintenance Phase

Case Study: Centenary Bank

A postgraduate dissertation report presented to the Faculty of Science in partial fulfilment of the requirements for the award of the degree Master of Science in Information Systems.

Uganda Martyrs University

MUGUME EDGAR

2014-MI32-20007

December 2016

DECLARATION

I have read the rules of Uganda Martyrs University on plagiarism and hereby state that this work is my own. Replace this page with the signed page.

DEDICATION

This work is dedicated to the Almighty God who has brought me thus far and Twine family for the continuous encouragement and support they have given me in regards to pursuing this Masters' Degree.

ACKNOWLEDGEMENTS

The success of this project was made possible because of the various contributions from the people surrounding me. It is in this regard therefore, that I express my gratitude to the various individuals and organizations, all in their respective capacities for their tireless efforts and role played towards the successful completion of my research project.

Thanks to the Project Manager – Exodus Project and Chief Manager for core banking, infrastructure at Centenary Bank, Mr. Ekemu Francis in particular, for having been able to provide a cordial research environment and ample interview time with his staff.

Special thanks owed to my supervisor, Mr. Mugejjera Emmanuel for the pivotal role played and pertinent guidance provided during the research period.

Finally, I am grateful to all my friends and workmates at Neptune Software whom I happened to interact with, and those who helped me in my research Olupot Douglas, Winnie Nabajju, Twesigye Wence, Bridget Nankanja, Andrew Babigaisa, the MIS class of 2014 and all those that were able to provide guidance and support, both spiritual, technical or otherwise, that I have not been able to mention, for all your efforts were integral components towards the successful completion of my research.

"May God Bless You All"

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Interface
GUI	Graphical User Interface
ICT	Information and Communications Technology
IT	Information Technology
MIS	Management Information Systems
SDLC	Software Development Life Cycle
UI	User Interface

ABSTRACT

Software issue tracking is basically the process by which an issue encountered by an end-user is reported to the software vendor/developer for fixing. A software Issue tracking system is an application that is designed to help in issue tracking, software quality assurance, and also for programmers to keep track of reported software issues in their work.

The core aim and significance of this research was to design and implement a fast, simple and effective issue tracking system with intelligent features such as automated ticket assignment, duplicate filtering, bug prediction and change control management, among others, hence delivering a high performance, lightweight and flexible issue tracking system called Octopus. The need for this arose when it was established and proved that existing issue tracking systems allow reporting of duplicate issues, lacked adequate features to allow automated assignment of issues hence making the issue management process manual and time consuming. These systems were also incapable of bug prediction and change control wasn't enriched. The researcher then concluded that having the above gaps addressed would significantly improve the timelines in which software issues are solved efficiently and effectively.

Using qualitative research methods to enable gathering of unknown software user needs and uncover all the variables surrounding issue tracking process workflows, the researcher deployed semi-structured interviews, and literature reviews. Also research strategies as Case Study Strategy and Design Science were used facts from this requirements gathering process where used to design of a fast, simple and effective issue tracking software with intelligent called Octopus. This system was thoroughly tested and evaluated by a team of experienced users that comprised of both software end users and software developers to prove that the gaps identified had been resolved.

In conclusion, issue tracking is an important part of every software project and using issue tracking systems is necessary, as it brings on board benefits such as improvement in the turnaround time of software issue resolution, increased user satisfaction and customer appreciation in software, better productivity of teams and reduction of operating expenses in software firms.

CHAPTER ONE

INTRODUCTION

1.0 Background

This research was conducted in the domain of Management Information Systems (MIS) specifically Project Management Information Systems.

Recent years have seen a massive growth and expansion of software in both open source and commercial sectors (Saldarini, 2010). Systems or their components are upgraded, fixed, or replaced, presumably to provide better functionality, ease usage, and reduction of operating expenses among others. This research mainly focuses on the life cycle of application software, particularly the operational and maintenance phase of software. Operations and Maintenance is the fourth phase of the Software Development Lifecycle. In this phase, systems are in place and operating. An operational system is periodically assessed to determine how the system can be made more effective, secure, and efficient. Often the need arises for enhancements or modifications to the system. These enhancements are then developed, tested, and added. After application of such enhancements or modifications, the system is monitored for continued performance in accordance with the enhancements made (Saldarini, 2010).

However, during this operational use of a given software, users encounter issues every now and then as they interact with the various functionality in the given system. These are primarily referred to as Software issues because at the point when the user reports this occurrence to the software developer/vendor, it is not yet clear or proven whether this particular occurrence in question is a System Bug / defect, or a missing functionality, or if it is just a mere user knowledge gap problem in line with the system usage. It is later after further analysis of the logged software issue that clarity is made on which category this issue falls and then a remedy is given appropriately. A missing functionality means that among the items scoped as deliverables for a given software by its vendor, this function in question is not supported by the software. Knowledge gap as often referred to as in the software arena is when a given functionality in a software is available and works perfectly well but for one reason or another, the user fails to navigate and use the function to obtain the desired results. A Bug/defect which has been noted to be the most delicate category

in Software issues is defined as a variance from specification causing unexpected behaviors in a given software. These bugs could have existed from the time of initial implementation of the software or may have been introduced during enhancements or modifications the software. However, regardless of the category, it is important that any Software issue as reported by a user is tracked and resolved diligently in a very systematic manner to ensure quick accurate delivery. This can best be achieved through the use of software Issue tracking system (Trajkov, 2011).

Software issue tracking is basically the process by which an issue encountered by an end-user is reported to the software vendor/developer for fixing. It can also be defined as process of tracking the life cycle of a software issue from the time it is encountered by the end-user and reported to the vendor/developer, till the time a fix is done and implemented to address the end users concerns. A software Issue tracking system is an application that is designed to help in issue tracking, software quality assurance, and also for programmers to keep track of reported software issues in their work.

Issue tracking systems in the software arena allow individual or groups of developers to keep track of outstanding bugs in their product effectively. Basically, a software Issue tracking system is a tool that facilitates management of issues in general and bugs in particular in a quicker manner and ensures higher quality management of software being developed or being used. Globally, these systems are widely used and they are treated as essential repositories that help in finding status of issues/bugs and quickly resolving them. Software engineers frequently use software issue/bug reports from these issue tracking systems to fix bugs as long as the reports contain enough information to point out the bug. This issue tracker software helps software teams manage issue reporting, assignment, tracking, resolution, and archiving via a reliable, shared to-do list that is used by numerous stakeholders throughout the lifecycle of the software and also serves as an archive of completed work (Dingsoyr and Royrvik, 2003).

A major component of an issue tracking system is a database that records facts about known bugs. These facts may include the time a bug was reported, its severity, the erroneous program behavior, and details on how to reproduce the bug; as well as the identity of the person who reported it and programmers who may be working on fixing it.

Typical issue tracking systems support the concept of the life cycle for a bug which is tracked through the status assigned to the bug. An Issue tracking system should allow administrators to configure permissions based on status, move the bug from one status to another, or delete the bug. Some systems will e-mail interested parties, such as the submitter and assigned programmers, when new records are added or the status changes.

However, most issue tracking systems are still lacking. Many of them are merely interfaces to a database that stores all reported bugs. As a result, they often ask too much from end-users who are not familiar with development practices. At the same time they cause frustration for developers who in turn get disappointed about the quality of bug reports submitted by users

In a survey conducted to software engineers of companies like Mozilla, Eclipse, and Apache found that the information items presented in bug reports play a very important role in fixing of bugs. It is further stated that, insufficient or improper reports caused delay in fixing bugs and thus causing crossing deadlines (Bettenburg et al, 2010). The information items that can be found in the bug reports include screenshots, test cases, expected behavior, observed behavior, stack traces and steps to reproduce the bug among others. Such information items is generally the minimum preferred requirement needed by engineers to fix bugs easily. However, the prior research in this area by (Bettenburg et al, 2010) and (Zimmermann et al, 2011) revealed that the bug reporters omitted these Essential information fields in their bug reports making them poorly designed reports. Such reports are of little use to developers for the purpose of fixing bugs. When developers need very descriptive information and the bug reports lack such information, it leads to stalling of the project or delay in completion with other cause and effects. Also, the previous works have not examined the ticket resolution process pace. A ticket simply refers to the receipt number given to a user after they have successfully logged an issue. Its main purpose is to provide a precise ID number to the user to ease the tracking of the issue on the issue tracking system. It is important to track the pace at which issues are being resolved, as large software development teams require a lot of labor and therefore a need to account for the efficiency of each team member.

Globally, some of the problems with the current issue tracking systems is that they have ignored crucial aspects such as; identifying cases when a duplicate of a previously encountered and resolved issue is being reported so as to stop it by probably enlightening the user that this issue was logged before, and going ahead to provide the already existing fix. This has proven to be time consuming and also wastes resources that could be applied elsewhere (Bettenburg, 2011). Currently when issues are logged by users, they wait on manual intervention by whoever assigns the issue to various developers at software vendor end. Ideally there should be features and mechanism that enable issues to automatically be assigned to developer for attention as soon as they are logged to avoid time wasting in the issue resolution process (Baysal et al, 2011). Change control also isn't enriched in the existing issue tracking systems and hence the ability to maintain precise and updated system documentation is insufficient (Zatul et al, 2012). Furthermore, issue/ bug prediction basing on the recently resolved issues is another good technique that would efficiently help in the quality assurance process as it will point the testers of the solution to check potential logical errors that may arise as a result of fixing a given bug or enhancing a given module (Lyu, 2011). Most existing issue tracking systems have however not taken into considering the mechanism of adequate bug prediction.

In Uganda, software issue tracking systems have only been fully embraced on big software project implementations in large companies for instance at Centenary Bank under the Core Banking system support and maintenance. However their usage is still very low in medium sized and small companies. One of the reasons for their low usage is lack of exposure to the benefits of having issue tracking systems in place. But common reason is simply because their software vendors still use rudimental / traditional methods of Application Support that does not provide a trail when addressing software defects, which is very dangerous when it comes to proper support, growth and improvement of a given software. For those few like centenary bank that have been exposed to issue tracking systems, they have faced a number of challenges similar to those faced globally as has been stated above. These challenges have heard a negative impact in support and growth of software, but most importantly led to loss of revenue through delayed / inefficient resolution of crucial software issues.

The core significance of this research is to provide considerations for the design of a fast, simple and effective issue tracking software with intelligent features such as automated ticket assignment and filtering duplicates among others, hence delivering a high performance lightweight and flexible issue tracking software called Octopus. While some of the considerations presented in later chapters may already be addressed by existing commercial tools, it is the interplay and tradeoffs that must be made among them that I argue for the need to be carefully scrutinized and enhanced to provide a comprehensive software issue tracking system.

1.2 Problem statement

Existing issue tracking systems allow reporting of duplicate issues, a previously encountered and resolved issue can be reported one or more times probably by different users without their knowledge, this is time consuming (Bettenburg, 2011). An Ideal issue tracking system should have automatic assignment facility by showing the list of probable fixers of this type of bug that will save the moderator's time in searching the developer. This future is however lacking (Baysal et al, 2011) hence making the issue assignment process manual and time consuming. Change control isn't enriched (Zatul et al, 2012) in the existing issue tracking systems and hence the ability to maintain precise and updated system documentation is insufficient. Furthermore, issue/ bug prediction basing on the recently resolved issues has not been taken into consideration. (Lyu, 2011)

1.3 Main Objective

The broad objective was to design and implement a software issue tracking system that will enable quick and efficient resolution of software issues by enhancing critical issue tracking features such as issue auto-assignment, duplicate issue filtering and change control among other.

1.3.1 Specific Objectives

The specific objectives of this project are:

I. To perform the comparative study and analysis of current issue tracking system, including review of literature so as to identify and understand the gaps

II. To identify the limitations in these issue tracking system in study basing on the Gaps.

III. To propose and develop a model of a software issue tracking system that addresses the limitations identified from the research.

IV. To implement a prototype of the model called Octopus.

V. To test the prototype.

1.4 Scope

The geographical scope of this was at Centenary Bank head office in Uganda, targeting the IT support officers under the department of Business technology as the users (Reporters) and their core banking system vendors as the developers. The system scope was to be improved the issue tracking process by addressing the gaps encountered by user and developers during resolution of issues of their Core Banking System software in its Operations and Maintenance phase.

1.5 Significance

This system plays a significant role in Software project environments for both the users (Reporters) and the developers in the following ways:

a) Prevention from duplicate reporting of already resolved bugs that addresses the aspect of saving time in issue resolution, which is key in an operational business environment.

b) Ability to automatically assign issues also saves time which is usually wasted when issues are just logged without a particular person tagged to resolve them.

c) Since change control is a very delicate matter when dealing with deployment of new patches and new requirements, this system will ensure it provides avenue to capture and store necessary information for future reference to the solutions given.

d) Graphical analysis and presentation in areas mentioned provides some sort of dash board where management is able to assess efficiency of their staff, the software, or even the software Providers. This is key in business management.

CHAPTER TWO

LITERATURE REVIEW

2.0 Introduction

This research presents background information on issue tracking and issue tracking systems in the area of software development and operational maintenance. It constituted an in depth discussion on already accomplished related works in regards to issue tracking, issue tracking systems and software development team coordination. Through the exploration of this existing literature, this chapter will introduce the niche addressed by this research.

2.1 What is issue tracking?

There are a number of issue tracking software in existence today. While all of these systems vary considerably in their interfaces and features, they all share the same core purposes and functionality as described below. If we consider the textbooks definition of issue tracking, we find descriptions such as the following:

Issue tracking, often called bug tracking (and sometimes request tracking), is the process of keeping track of open issues in software development. Bug tracking is a misleading term in many ways and obviously depends on your definition of a bug. "Issue" is a broad enough term to describe most of the kinds of tasks you might need to track when developing and maintaining software, and so drives our choice of terminology here (Henderson, 2006).

Functionally, most issue tracking systems provide a form that allows us to report and manage bugs. They also provide a set of stored reports and graphs that allow us to analyse, manipulate and output bug data in various ways, and a customized workflow or life cycle that provides for orderly bug management. (Black, 2002)

Issue trackers, bug trackers, modification request control systems regardless of what specific terminology is used all roughly refer to the same class of software systems. In the most general of terms, these systems are designed to manage electronic artefacts that move from a starting state to an end state (and possibly visit several in-between states along the way), and accumulate information during their transitions between these states. More specifically, issue tracking systems are databases that keep track of bodies of information, outstanding issues such as software defects or "bugs," feature requests, customer inquiries, etc. The variety in the types of information stored in issue tracking systems has been acknowledged since their earliest incarnations.

A Modification Request (MR) still applies over 30 years after it was started. A Modification Request (MR) is a request to those in charge of a system to modify that system (Jaantti et al,2012). The request might be to add, modify, or delete capabilities, to fix a bug, to issue a new version of a system, or even to create a system. Thus our definition of an MR includes trouble reports, design change requests, enhancement requests, etc. A request may result in changes to system hardware, software, or documentation or it may result in no changes at all (Jaantti et al.2012).

This information represents tasks that move from an opening state to a closed state over time. Along the way, these information artefacts often accumulate additional information as various people within the software team work on them. For example, a bug might contain instructions on how to reproduce the problem, or a feature request might be refined with additional specifications or sketches. Each of these items (or "modification requests" or "issues") can also have certain attributes associated with them. A bug, for example, might include things like the time the issue was filed, who filed it, what version of the software the bug applied to, and the severity of the bug (e.g., a critical bug that causes loss of data versus a minor one such as an aesthetic imperfection).

The primary interest lies in issue tracking systems in the software development and maintenance domain. Given the relatively free and unconstrained usage of the term *issue tracking*, there are also similar systems that this research will not be examining in this thesis—in particular, helpdesk

trouble ticket trackers, and business issue management systems. While these systems do represent a common manifestation of *issue tracking*, their primary purpose is not targeted toward software development and are thus outside the focus of this research.

2.2 Generic Process of the Issue / Bug reporting in Issue tracking

The issue reporting/tracking system provides an interactive web based platform for bug reporting and progress tracking. The system may involve a generic process or specific schedule of bug reporting process also known as a change request. Common items of interest captured during bug reporting include;

a) Reporter's name or Id.
b) Type of change request: Whether it's a bug, enhancement or a new requirement.
c) Description: Detailed description of the bug including what, where, why, how and when the bug occurs. Actual message that appears during the operation may be included with the actual set of input and expected output.
d) Version or Build number: Specify the version of the project.
e) Component: Specific component need to be specified.
f) Screenshot/Attachment: Corresponding screenshot can also be uploaded as a .jpg or .gif file by capturing the actual operation/output/message.
g) Priority: Priority may be assigned for its urgency.
h) Severity: Specify frequent occurrence and its impact in the system.
i) Status: Current status of the bug (new, opened, confirmed, closed, etc.).
j) Created by: Name of the person or id already registered with the system who is reporting the bug.
k) Assigned to: The bug reporter may also assign bug to specific person, if one knows about a particular person who can solve this problem otherwise assigned by the moderator.
l) Revision History: If the bug is earlier reported, show the historical changes.
m) Estimated time: The estimated time may be specified, generally used in case of closed team environment not in the open source environment.
n) Comments: Any other information that is helpful in identifying the bug.

Within an issue tracking system, each item (or issue or case) generally follows one of a few predefined paths from when it is first opened until it is eventually closed. Depending on the type of issue, there may be various intermediate states as well as cycles within its path. Each stage in the life of an issue is characterized by a status, or state. The number of states may vary from system to system. In addition, within a single issue tracking system, different issues will go through a different succession of states (Jaantti et al, 2012).

This set of paths is often referred to as the *workflow* supported by an issue tracking system (Bugzilla, 2009). Workflows can vary from very simple and open-ended to very detailed and complex. The state diagram in Figure 2.1 illustrates a minimal set of states required for an issue tracking system's workflow, namely, the *open* and *closed* states and the transitions between them. When a new bug or feature request is created it starts in the *open* state. After work has been completed on this item it follows the *resolve* transition to end up in the *closed* state. If the issue is later found to be incomplete (e.g., a bug reappears or a feature is not fully implemented) it can follow the *reopen* transition back to the *open* state.

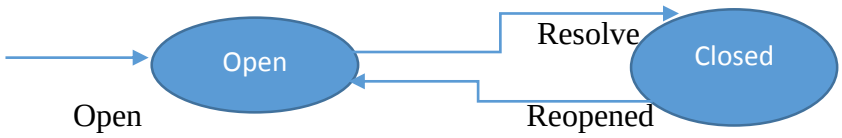

Figure 2.1: Simplified issue lifecycle

The above process looks simple but there are a number of situations in which such a simplified view of issue tracking falls apart. For instance, many software development teams require issues to be verified before officially being marked as closed: When a bug is resolved, it gets assigned back to the person who opened it. This is a crucial point. It does not go away just because a programmer thinks it should. The golden rule is that only the person who opened the bug can close the bug. The programmer can resolve the bug, meaning, "*hey, I think this is done*," but to actually close the bug and get it off the books, the original person who opened it needs to confirm that it was actually fixed or agree that it shouldn't be fixed for some reason. (Fog Creek Software 2009b).

The system produced from this research (Octopus) shares a similar architecture to The Bug Track as shown in figure 2.2 below which aims to be as simple enough for small teams of development yet still effective for the job.

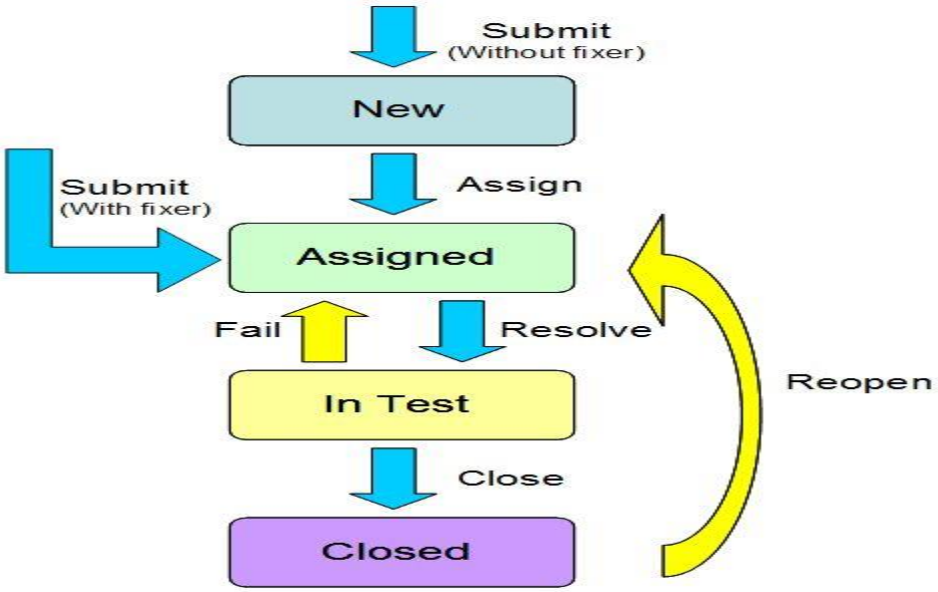

Figure 2.2: Generic bug life cycle flow

Most bug tracking systems currently availed also tend to lean towards a generic bug life cycle flow **(figure 2.2)** can be described as follows.

a) When the bug is first reported, its status is usually set to *"Unconfirmed"* as in the case of Bugzilla. Bugzilla is one of the popular issue tracking system and it was brought to existence during the Mozilla Firefox project to help track issue surrounding the development and implementation of the Mozilla Firefox product. After its first review, the status can be updated to *"New"* or "*Assigned* based on the previous bug history databases.
b) Once the assignee has corrected the bug, he or she can update its status as *"Resolved"* by specifying how it was resolved i.e. "*Fixed*", "*invalid (not a bug)",* "*duplicate*", "*won't fix*", and "*works for me*". A resolved bug, will not be "closed" until someone else involved confirms it out or it will be confirmed by the moderator.

c) Once it is confirmed, the bug status becomes "verified" otherwise the status is set to "*Reopen*". It remains in "*Verified*" state until it is incorporated in the release version of the product and its status will be updated to "*Closed*".

d) The closed bug can be reopened with its status as "New" or "Unconfirmed". There might be many bugs that are currently being fixed and some of bugs that are waiting for resolution set to be "closed".

e) A status report can be generated from time to time with number of bugs pending in each category as a status and emailed to the assignee as well as submitter of the bug. So the moderator or assignee can see the workload on individual and may postpone the bugs for some time or reassign it to others for its resolution. In this way, the bug can be effectively resolved in an efficient way. These status reports are also crucial in this process as they greatly assist in tracking progress. Most bug tracking systems currently have these in place and each system avails them with varying extent of details. These reports can also be summarised into an information dashboard for consumption of higher Management who may lack time to analyse report by report in order to establish the level of efficiency. As we shall see later on in this literature review, most issue tracking systems have neglected the need to have a comprehensive dash board to serve this purpose.

2.3 Issue tracking systems classification criteria

The issue/bug tracking system vary from general purpose to the specific purpose. The generic system will have many features while the specific purpose systems may have limited features. A suitable issue tracking system can be selected based on user's requirement.The user's requirement generally varies from the platform, support, size, reporting, tracking and other features of the project. These broad criteria can be further classified or categorized in three or more sub-categories as given in **table 1**.

Platform: refers to license policy, architecture, operating system, web server, back-end database, programming language and clients.

Support: refers to language, multiple projects, web interface, database, email notification and localization.

Size and usage: refer to the size of the project, number of downloads, number of releases and number of clients/usage.

Reporting: refers to how issues are reported that includes; forms incorporated, email, attachments, web and feedback to the users/reporters.

Tracking: refers to assignment of issues, timely update status of the bug, graphs for the number of pending bugs, assigned bugs, and bugs fixed over a period of time in each category, bug prediction, duplicate bug filtering and search facility for new as well as existing bugs to know about their status.

The other feature refers to any other feature that is not a part of the above mentioned categories.

A summary table showing Classification categories for Bug Tracking Systems (Table.1)

Platform	**Support**	**Size and Usage**	**Reporting**	**Tracking**	**Other Features**
License	Language Support	Size of the project	Form based	Timely updates	Any other features that may not come up an specific category
Architecture	Multiple projects	Number of downloads	Email based	Graphical display/reporting	
Operating system	Web interfaces	Number of releases	Attachments	Search facility	
Web-sever	Databases	Number of clients/ usage	Web (Online)	Issue Assignment	
Back-end database	Email Notifications		Feedback	Bug Prediction	
Programming language	Localization			Duplicate bug filtering	
Client				change control	

Table 2.1: A summary table showing Classification categories for Bug Tracking Systems

2.4 Comparative Analysis of existing issue tracking systems

Plenty of issue tracking systems are available in the industry. We chose the most popular, most used and also still under improvement process. The features and functionalities in these tools were deeply analysed basically to establish and acknowledge the already done works and also point out those works that need improvement. The tools considered in this analytical review are;

a) Bugzilla,
b) Jira,
c) Trac,
d) Mantis,
e) Gnats,
f) BugTracker.Net,
g) Fossil.

The comparison analysis on the following issue tracking systems above was carried out basing on the issue tracking systems classification criteria mentioned earlier above in 2.3.

2.4.1. Classification Based on Platform Characteristics

The comparison criteria based on platform characteristics are shown in table 2 below. In this table, most of these tools are available in open source environment. Mantis is closed source while its free version is available for download and use. Some of the tools are available as a licensed version for which support can be asked from the ownerPaid version of Jira comes in three different variants: such as Standard, Professional and Enterprise based on the size of the project and support. Most of these tools work in web based environment while their early versions were available as client/server based environment.

Trac is developed on wiki based architecture, i.e. anyone can see bugs, fix bugs and edit any part of it. Operating environment for most of the projects are cross-platform, i.e. they can be installed on any machine.

BugZilla and GNats are available on Linux while BugTracker.Net is available on windows environment. The clients are browser based, so they are independent of the platform. For web server, most of the tools required apache/tomcat except BugTracker.Net which requires MS-IIS. Most of these tools support MySQL as a backend database except BugTracker.net which requires MS-SQL Server and Fossil requires SQLite. These tools may vary on the basis of programming language used in coding. Tools are developed in C, Python, and PHP, Perl, Java and C #.Net programming languages.

Comparative Analysis of Features Based on Platform Characteristics (Table 2)

Tools / Platform	Bug Zilla	Jira	Trac	Mantis	Gnats	BugTracker.Net	Fossil
license	Open Source/Free/ Proprietary	Free for non-commercial use/ Proprietary	Open Source/ Free	Free/ paid	Free and Open Source	Open Source	Open Source/Free
System Architecture	Client server / Web based	Client server / Web based	Web based/ Wiki Based	Web based/ WAP	Web based	Web based	Distributed Web based
Server Operating System	Linus	Cross-platform	Cross-platform	Windows, Linus, Mac-OS	Linux	Windows	Windows, Linus, Mac-OS
Web server	Apache, MS-IIS, or server capable to run CGI	Apache Tomcat	Apache	Apache and MS-IIS	Apache	MS-IIS	Apache and MS-IIS

Backend Database	MySQL, Oracle and PostgreSQL	Mostly supports all RDBMS	MySQL, SQLlite, Postgre-SQL	MySQL, MS SQL, and PostgreSQL	MySQL	MS-SQL Server	SQLLite
Programming languages	TCL/Perl	Java	Python	PHP	C	ASP.Net	C
Client (Web Browser)	Anyone	Anyone	Anyone	Anyone	Anyone	Anyone	Anyone

Table 2.2: Comparative Analysis of Features Based on Platform Characteristics

2.4.2 Classification Based on Support Characteristics

The important component for comparison of tools is support features which are shown in **table 3**. The general characteristics for support features are Language, Web Interface, Maintenance support, Email notification and Localization. Most of the tools support multiple languages based on Unicode system. Only Fossil and GNats do not have the support of multiple language features. Again in the same line, most of the tools support web based interface. People are more fascinated to use browser based interface because it is independent of operating system environment.

Fossil does not have the complete web based interface. Maintenance support is also available for free of cost or with a paid service with nominal charges. Whenever, a new bug or update is reported, an email will be automatically sent to all users who are in the registered mailing list with the project. The email will also be sent to all registered users even when there is a small update.

Bug localization, i.e. finding the most relevant part of source file hierarchy of software in the bug repository database is another important feature provided by BugZilla, Trac and Mantis while others does not have.

Table 3: Comparative Analysis Based on Support Characteristics

Tool / Support	BugZilla	Jira	Trac	Mantis	Gnats	BugTracker.Net	Fossil
Language Support	Yes	Yes	Yes	Yes	No	Yes	No
Web Interface	Yes	Yes	Yes	Yes	Yes	Yes	No
Maintenance Support	Yes	Yes	Yes	Yes	No	Yes	Yes
Email Notification	Yes	Yes	Yes	Yes	Yes	Yes	No
Bug Localization	Yes	No	Yes	Yes	No	No	No

Table 2.3: Comparative Analysis Based on Support Characteristics

2.4.3 Classification Based on Size and Usage Characteristics

Another component of the feature to compare is the size/usage of the tools. In this category, size of the project, support of multiple projects, number of downloads, number of releases and clients per usage are the important parameters for which the statistics can be collected over different time period that may be from the date of first release to the recent date of release. The statistics were collected by browsing the corresponding project website wherever it is available as shown in the table 4.

Table 4: Year of First and Latest Release

Tool / Release Status	BugZilla	Jira	Trac	Mantis	Gnats	BugTracker.Net	Fossil
First Release	1998	2004	2006	2000	1992	2002	2006
Latest Release	2011	2013	2010	2013	2005	2011	2012

Table 2.4: Year of First and Latest Release

Looking at the table, conclusion can be drawn about the current development status of these trackers, i.e. whether it is active or dead. It will also inform about the life of trackers that will be helpful, regarding its integrity and maturity. Here, it is clearly shown that there are many releases for the trackers Jira, Fossil and BugZilla. The reason may be the frequent releases which occur and early fixes are being incorporated and updated version is being made available to the users

2.4.4 Classification Based on Reporting Characteristics

Another important component is reporting of bug to the project. This can be done by either of these methods, i.e. form based, email, attachments, web (online) and feedback. This is shown in table 5.

Table 5: Classification Based on Reporting Characteristics

Tool / Reporting	BugZilla	Jira	Trac	Mantis	Gnats	BugTracker.Net	Fossil
Form Based	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Email	Yes	Yes	No	Yes	Yes	Yes	No
Attachments	Yes	Yes	No	Yes	No	Yes	No
Web(Online)	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Feedback	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Table 2.4: Classification Based on Reporting Characteristics

A well designed reporting feature attracts users in submitting the bug effectively.

2.4.5 Classification Based on Tracking Characteristics

The tracking feature contains subcategories like; issue assignment, timely update status of the bug, graphs for the number of pending bugs, change control, assigned bugs, and bugs fixed over a period of time in each category, bug prediction, duplicate bug filtering and search facility for new as well as existing bugs. Most of the systems have one or more of these functionalities however some are generally lacking in most as shown in table 6.

Whenever new updates are received, the registered members receives an email regarding the updates as well as new release of the system. The graphical and charting facility is one of the very important features that tell user about the number of pending bugs, number of open bugs, number of closed bugs, time taken by each bugs in the form of interactive charts and estimated time to resolve the bug. To track the bug, the tools have the facility to search for a bug that is already submitted in the system. This is helpful in identification of duplicate bugs (Bettenburg, N, 2011)

Current day's product must have all these facility with dynamic graphical and charting options. The search facility is also one of the important considerations. Generally, the systems have full text search facility on title, description and summary of the bug report.

Table 6: Classification Based on Tracking Characteristics

Tool / Tracking	BugZilla	Jira	Trac	Mantis	Gnats	BugTracker.Net	Fossil
Timely Updates	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Graphical Display / Reporting	Yes	Yes	No	Yes	No	Yes	Yes
Search Facility	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Auto Issue assignment	No	No	No	No	No	No	No
Duplicate bug filtering	No	No	No	(Yes)To some extent	No	No	(Yes)To some extent
Bug Prediction	(Yes)To some extent	No	No	No	No	No	No
Change control	Yes	Yes	No	No	No	Yes	No

Table 2.5: Classification Based on Tracking Characteristics

Some of the systems are able to provide the search facility on any of the parameters such as bug id, severity, priority, time of submission, author, assignee, number of reassignment/history of assignment, etc. There are number of parameters on which the search facility can be provided. Jira has the limited parameters search options but powerful. Mantis provides the bar chart based result of the search to show the progress of a particular bug.

As shown in table 6, some of the systems have the capability to offer features such as; automatic ticket assignment, bug prediction, duplicate bug filtering and change control however these functionalities are not mature.

2.4.6 Classification Based on Other Features

Any other feature may incorporate many more features that are very important but may not have one special category. These are browsing of source code repository, version control, and Subversion facility. Some of the above mentioned systems have the capability of these features but they are not mature. Overall the above mentioned subcategories can be further sub divided into specific subgroups. It will enable to provide a better picture of the bug reporting and tracking

2.5 Qualities and capabilities of a Modern Bug/Issue Tracking System

According to (Singh et al 2011), way of bug reporting is the most crucial and initial first step towards having a bug resolved quickly and correctly. They further outline the following below as guidelines to good reporting.

a) The system has all the ways and means of reporting the bug, i.e. by using minimum field form, email support, attachments, web based and feedbacks, etc. It's capable of asking minimal requirements from the user and generate automatically most of the parameter's values using mathematical models, reliability models, statistical models and machine learning techniques. This will encourage the user to report the bug.

b) It should be capable of automatic bug assignment by showing the list of probable fixers of this type of bug that will save the moderator's time in searching the developer. (Baysal et al , 2012)

c) It should be able to identify duplicate bug entries and reassign the bug to the respective handler otherwise provide an already implemented solution.

d) It's capable of providing the LDAP support to enable single sign-on process. The status of the user from reporter to developer and developer to moderator can be updated based on his previous activities. Single user-id sign-on must be used to report/fix/monitor the bug in/from the system.

e) User profiling can be done on the basis of their bug removal efficiency. The threshold values of number of submitted bugs, number of assigned bugs and number of removed/fixed bugs by a particular user should be able to put the user in the privileged category of the user. The privileged category of user is the active and reliable user who is continuously contributing in the development of the system. It must have the database of all the developers who are useful for fixing of the future bugs. It means that if the bug/fix is submitted by these groups of user, this should be accepted immediately and requires minimal interventions in incorporating their solutions in the next build or release.

f) The system provides the exact classification of errors/bugs/patches/features using the past report submitted. The classifiers can be developed by using the supervised classification techniques.

g) It should be a web based interface as mentioned earlier and is available on a 24/7 basis supporting reporting at any point of time.

h) It should be fully integrated with version control and able to communicate with major version control system available. This will be helpful in browsing in source code history, predicting source code change and future bug prediction using suitable data mining or statistical techniques.

i) It should provide SOAP/Restful web services for integration with the other system. Support of web services helpful in sharing the data from the diverse platform.

j) It also comes with an Android application for users to monitor and track the progress of their logged defects.

Tracking the progress of fixing the bug process / reliability assessment requires the following in the system:

a) It is capable of providing the current status of the bug in a graphical way. Status of the bug can be depicted using some color coding scheme. For example, when the bug is first submitted, the color of status can be set as Red, if the bug is assigned and in progress; the color of status can set as yellow. If the bug is fixed, the color of status will be set to green. The color coding scheme can use some more colors based on the value of status. A very powerful phrase says "Picture speaks thousands of words", It means by looking at the color of status, the state of the bug can be decided very easily.

b) It is capable of estimating the time taken by the developer to resolve the bug. The system must be able to calculate the time to resolve bug using the history of related bugs or developer fixing the bug unless otherwise.

c) It is capable of generating timeline graphs for time taken and remaining time required to fix the bug. By looking at the charts, it can be easily figured out that when moderator can plan for the next release.

d) Mathematical models can be incorporated in the system that will predict the number of bugs yet to be fixed and their complexity. On the basis of reported bugs, the priority and severity can be classified from developer's perspective. The complexity of the bugs and their distribution will be helpful in determining the optimal effort required in assigning and fixing of bug.

e) Imperfect debugging and error generation can also be incorporated along with the bug prediction because there may be chances of introduction of some new bugs during bug fixing or the bug may not be perfectly fixed due to incompetency/lack of complete knowledge about the bugs.

f) Software reliability growth models (Singh et al, 2010)], other statistical and machine learning models (Lyu, 2011) may be incorporated in the bug prediction. Bug predicting tool, which may be an integral part of the bug racking and assessment system, will able to

provide the useful information to the active user or developer in making plans to remove of bugs, tracking, and releases, etc.

g) It is capable of identifying the module for which bug/fix is related in a graphical way. It should have the capability of source code visualization and co-change prediction facility. The graphic based source code browser will be helpful in co-change prediction.

h) It is capable of managing the change management system, build release management system, and intelligently automatic reporting system whenever there is some submission.

i) Automatic email notification whenever there is any significant updates or submission of reports. It enables the users to get updates that are fixed recently.

j) It is capable of supporting the inclusion of test simulations in the system itself. It enables the system to check for the left out bugs due to this fix.

k) Role based user interfaces should be provided based on access level of the user.

l) Intelligent search facility should be provided in the system which will be helpful in detecting the duplicate bugs or related bugs.

m) It is capable of providing a graphical view of number of clients, number of releases, number of downloads, current as well past build releases with respect to time. It should also be able to calculate some of the derived parameters behind the box and display on the user's screen to facilitate the user regarding the role and value of those parameters.

n) Documentation of the bug fix capability for the developer/resolver to add when updating the logged issue.

The above mentioned process/features if included in the system can make reporting, assignment and reliability assessment become quite easier.

2.6 Issue reporting best practices

Whenever a user stumbles into a problem, they should be able to create a new issue request (ticket). In a well-written report, one is required to deliver the following in the report: (Hauner, 2008)

a) Explain how to reproduce the problem;
b) Analyze the error so you can describe it in a minimum number of steps;
c) Include all the steps;
d) Make the report easy to understand;
e) Keep your tone neutral and non-antagonistic;
f) Keep it simple: one bug per report; and
g) If a sample test file is essential to reproducing a problem, reference it and attach it.

As was already stated, there are many ways of how a request can end "in the trash" instead of being addressed. To raise the chance of an issue being fixed, there are couple rules which should be followed.

Rule #1 – Write good summary

This one-line description of the problem is the most important part of the report.

a) The project manager will use it in when reviewing the list of bugs that have not been fixed.
b) Executives will read it when reviewing the list of bugs that will not be fixed. They might only spend additional time on bugs with "interesting" summaries.
c) The ideal summary gives the reader enough information to help decide whether to ask for more information. It should include:
    i. A brief description that is specific enough that the reader can visualize the failure;
    ii. A brief indication of the limits or dependencies of the bug (how narrow or broad are the circumstances involved in this bug?); and
    iii. Some other indication of the severity (not a rating, but helping the reader envision the consequences of the bug).

Rule #2 – Explain how to reproduce the problem

a) First, describe the problem. What is the bug? Do not rely on the summary to do this.
b) Next, go through the steps that you used to recreate this bug.
    i. Start from a known place (e.g. boot the program);
    ii. Then describe each step until you hit the bug;
    iii. NUMBER THE STEPS. Take it one step at a time.
c) If anything interesting happens along the way, describe it. (One must give people directions to a bug. Especially in long reports, people need landmarks.)
d) Describe the erroneous behaviour and, if necessary, explain what should have happened. (Why is this a bug? Be clear.)
e) List the environmental variables (configuration, etc.) that are not covered elsewhere in the bug tracking form.
f) If the reader may have trouble reproducing the bug (special circumstances are required), be clear about these circumstances.
g) Keep the description focused:
    i. The first part of the description should be the shortest step-by-step statement of how to get to the problem.
h) Add "Notes" after the description, if there are any. Typical notes include:
    i. Comment that the bug will not show up if step X is executed between step Y and step Z.
    ii. Explain the reasoning for running this test.
    iii. Explain why this is an interesting bug.
    iv. Describe other variations of the bug.

Rule #3 – If two failures are visible, write two reports

Combining failures on one report creates problems:

a) The summary description is typically vague. Words like "fails" or "doesn't work" are used instead of describing the failure more vividly. This weakens the impact of the summary.
b) The detailed report is typically lengthened. Bug reports should not read something like the following: “Do this until that happens, in which case do not do this until the first thing is

completed and then the test case of the second part must also be complete, and sometimes you may see this, but if not then that…."

c) Even if the detailed report is rationally organized, it is longer (there are two failures and two sets of conditions, even if they are related) and; therefore, more intimidating.
d) Often, one bug gets fixed, but not the other.
e) When reporting related problems on separate reports, it is a courtesy to cross-reference them.

Rule #4 – Eliminate unnecessary steps

Sometimes, it is not immediately obvious which steps can be dropped from a long sequence of steps in a bug. Look for critical steps. Sometimes, the first symptoms of an error are subtle.

A list now exists of all the steps that were taken to show the error. Now, the aim is to shorten the list. As each step is executed, any hints of errors must be examined. The following factors should be examined:

a) Error messages (i.e. a message appeared 10 minutes ago. The program did not fully recover from the error and the problem evident now is caused by the poor recovery.)
b) Delays or unexpectedly fast responses.
c) Display oddities, such as flashes, repainted screens, a cursor that jumps back and forth, multiple cursors, misaligned text, slightly distorted graphics, doubled characters, omitted characters, or display droppings (pixels that are still coloured even though the character or graphic that contained them was erased or moved).
d) Sometimes, the first indicator that the system is working differently is that it sounds a little different than normal.
e) An in-use light or other indicator shows that a device is in use when nothing is being sent to it (or a light that is off when it should not be).
f) Debug messages: the debug monitor should be turned on (if it exists on the system) and should be monitored if or when a message is sent to it.

If, what seems like a critical step has been encountered, everything else from the bug report should be eliminated. One must now go directly from that step to the last one (or few) that shows the bug. If this does not work, individual steps or small groups of steps can be removed.

Rule #5 – Variations after the main report

The failure may look different under slightly different circumstances. For example:

a) The timing changes if additional two sub-tasks are performed before the final reproduction step is hit.
b) The failure will not show up at all or is much less serious if something else is put at a specific place on the screen
c) The printer prints different characters (instead of the characters described) if the file is made a few bytes longer

This is all useful information for the programmer and it should be included. But to make the report clear:

a) It should start with a simple, step-by-step description of the shortest series of steps needed to produce the failure.
b) The failure must be identified. (Descriptions of how it looks like or what impact it will have.)
c) A section should be added that states "ADDITIONAL CONDITIONS" that describes, one by one, the additional variations and the effect on the observed failure.

2.7 Software Development Life Cycle (SDLC) Model

According to (Hoffer, 2011), a Software Development Life Cycle Model is a conceptual model used in project management that describes the stages involved in an information system development project, from an initial feasibility study through maintenance of the completed application.

The following phases as in (Figure 2.3) are generally present in each and every software development life cycle model:

i. Identifying problems, planning.
ii. Analyzing the system needs
iii. Designing the system
iv. Developing and documenting the Software
v. Testing the System
vi. Implementing and maintenance.

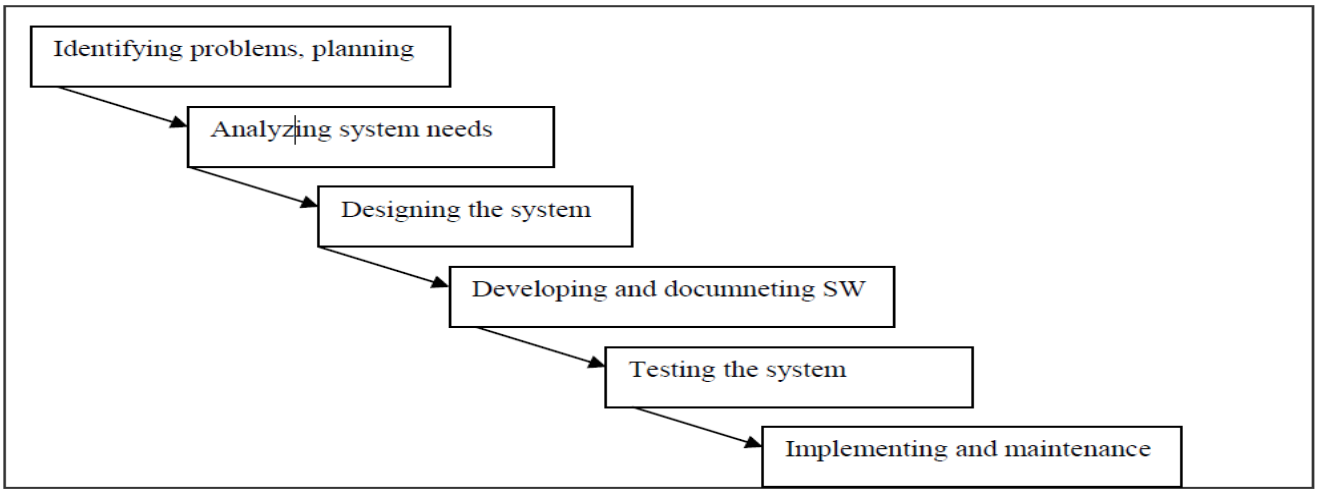

Figure 2.3: Common phases of an SDLC model (Source: Hoffer, 2011)

There are several SDLC models and these include the following:

i. Waterfall Model
ii. V-Shaped Model
iii. Evolutionary Prototyping Model
iv. Spiral Method (SDM)
v. Iterative and Incremental Method
vi. Extreme programming (Agile development)

According to (Hoffer, 2011), there are a number of SDLC models with each SDLC model having its own advantages and disadvantages. Articles by (Breu et al, 2010) explain steps which are best recommended when developing issue tracking systems. These steps are best aligned with Evolutionary Prototyping Model.

In Evolutionary Prototyping Model, the prototypes evolve into the final system through iterative incorporation of user feedback.

Below (Figure 2.4) is an illustration of the proposed methodology for this project. (Breu et al, 2010).

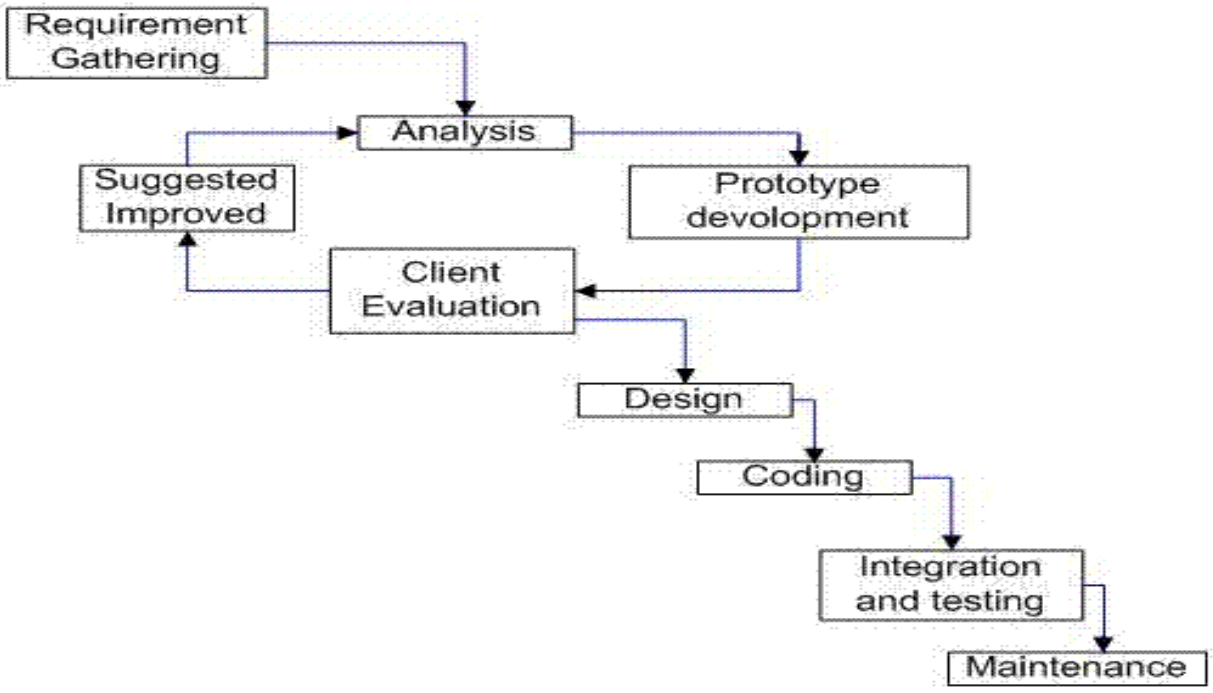

Figure 2.4: Evolutionary Prototyping Methodology (Breu et al, 2010)

The advantages of Evolutionary Prototyping Model

Since it is a series of repetitive iterations, it will be easy for customers to see some developments. If this is the case, gaining a positive impression from the target market is attainable because they can see how well you have been looking out towards improving systems to better handle customer service.

Additionally, when the current systems are complicated and ineffective, the Evolutionary Prototyping Model will become invaluable since it can pave the way for better systems through a series of iterations. The project is only considered complete once there is already the creation of the perfect system.

The disadvantages of the Evolutionary Prototyping Model

With this particular model, it may be very difficult to forecast the completion date of the project. It is a continuous development, set up by as many iterations or prototyping as possible that it becomes hard to tell when it will be complete. For as long as there are still significant errors in the system, it will undergo further iterations. This can cost time and money for the company.

It is also important to note that since it is a trial-and-error process, it would require a good project management team in terms of knowledge and skills so as to make them capable of countering surprises along the way. Moreover, as an ever-changing process, getting dynamic and flexible people to join would be a very good advice.

CHAPTER THREE

METHODOLOGY

3.1 Introduction

This chapter contains the methodology the researcher considered during this research and provides a discussion on the various approaches that were employed to achieve the stated specific objectives of the project. It includes the data collection strategy, the system design, how the system was implemented and tested and in order to meet the object of the research. This research project utilized the Evolutionary prototyping design methodology where the prototypes evolve into the final system through iterative incorporation of user feedback. The reason this design methodology was chosen among others is explained in Chapter 2 under section 2.7.

3.2 Research Design

This section describes the research approach and strategies, why they were chosen and how they have been used to achieve the research objectives.

3.2.1 Research Approach

The researcher used a Qualitative research design approach briefly described below.

3.2.1.1 Qualitative Approach

Qualitative Research is primarily exploratory research. Qualitative research is used to gain an understanding of underlying reasons, opinions, and motivations. It provides insights into the problem or helps to develop ideas or hypotheses for potential quantitative research.

Qualitative data collection methods include unstructured/semi-structured techniques such as focus group discussions, individual interviews, and participation/observations. The sample size is typically small, and respondents are selectively involved.

The researcher used qualitative approach because of its ability to enable gathering both unknown software user needs and uncover all the variables surrounding issue tracking process workflows.

The qualitative methods deployed by the researcher included semi-structured interviews, and literature reviews. Please refer to section 3.3.1 for the details of the specific steps for these methods.

3.2.2 Research Strategies

The research strategies used by the researcher included the Case Study Strategy and Design Science.

3.2.2.1 The Case Study Strategy

Case study research strategy is an empirical inquiry that investigates a phenomenon within its real-life context. Case study research can be done on a single and/or multiple case studies, can include both qualitative and quantitative evidence, and relies on multiple sources of evidence (Yin, 2003).

The case study research strategy proposes techniques for organizing and conducting the research successfully and it includes six steps: Determine and define the research questions, select the cases and determine data gathering and analysis techniques, prepare to collect the data, collect data in the field, evaluate and analyze the data, and prepare the report.

The case study research approach was deployed by the researcher to gather an understanding of technicalities involved in software issue tracking and management. The six steps of the case study strategy is applied to study multiple entities, processes, activities, inputs, outputs, and actions of software issue tracking and management in the following ways:

i. Determine and Define the Research Questions. In general, a software issue tracking system involves multiple variables such as entity types, activities, inputs, outputs, and actions. In this case, the researcher is primarily interested in determining all the variables, relationships and constraints that crosscut in software issue tracking and management. After reviewing the literature surrounding software issue tracking, the following questions for the study were used:
    a. How effective is the software issue reporting process and what impact does it have on software issue resolution timelines and software development as a whole.

b. What role does lightweight communication (email, chat, instant messaging, wikis, twitter, verbal conversation, etc.) play in the day-to-day completion of software development tasks?
c. How does such communication apply to, interoperate with, and sprout from issue tracking tasks (filing software bugs, working on feature requests, handling customer support problems, etc.)?
d. How might this communication and decision making when performing ticket assignment be better leveraged and/or supported by software development tools?

ii. Select the Cases and Determine Data Gathering and Analysis Techniques. Refer to section 3.3.1 below for detailed data gathering and analysis techniques applied by the researcher on the case study.
iii. Prepare to Collect the Data. Refer to section 3.3.1 below for the preparatory activities before data collection was undertaken.
iv. Collect Data in the Field. Refer to section 3.3.1 below for the interview questions prepared from the defined research questions above.
v. Evaluate and Analyze the Data. The researcher studies all the software issue tracking and management methods to identify all the entities, variables, actions, input and output within the data for all the processes.
vi. Prepare the Report. The output of this phase is a software issue tracking and management requirement document.

3.2.2.1.2 Description of the Case

Centenary Rural Development Bank Ltd started as an initiative of the Uganda National Lay Apostolate. In 1983 as a credit Trust and later began operations in 1985 with the main objective of serving the rural poor and contributing to the overall economic development of the country. In 1993, Centenary Rural Development Bank Ltd was registered as a full service commercial Bank. Today they are the leading Microfinance Commercial Bank in Uganda serving over 1,400,000 customers. Centenary Rural Development Bank services can be accessed across our 65 branches, 167 ATMs and the phone banking (CenteMobile) platform.

Centenary Bank uses a number of financial systems, to perform its day to day transaction processing tasks and banking automations. Among these systems is a core banking system called Equinox and is very key in automation of its banking and financial activities. This case study was chosen in regards to how the operational and maintenance phase of Equinox core banking system is done. Since Equinox is a big system, its users encounter issues from time to time. Equinox system users range from; accountants in back office operations, Tellers in the front office, IT staff who are responsible for user support, configurations, maintenance of the system (both Application and database). When Equinox users (Tellers, accountants, IT officers and others) encounter software related issues as they interact with the system, they contact the banking system support staff under the IT department who then assist them with in resolving the issue. If this software issue cannot be achieved, the banking system support staff then log it onto JIRA. JIRA is a software Issue tracking system that Centenary bank's banking system support staff uses to communicate software issue to their Core banking System vendor, so as to get assistance in resolving the software issue. Response and collaboration is after them carried out on JIRA until the issue is achieved and closed after application of the solution. In this research, the researcher used this case study in finding out how Centenary bank's banking system support staff currently use JIRA in issue tracking, the challenges they face , and how they feel that these challenges can be addressed in order to better issue tracking and resolution timelines.

3.2.2.2 Design Science

Design Science is a research methodology that offers specific guidelines for evaluation and iteration within Information Systems related research projects. The design science research methodology (DSRM) incorporates principles, practices, and procedures required to carry out research and meet three objectives: it is consistent with prior literature, it provides a nominal process model for doing DS research, and it provides a mental model for presenting and evaluating Design Science research in Information Systems (Roland et al , 2011).

Design Science research involves the creation of knowledge through the design of artefacts and analysis of the use of such artefacts in order to improve understanding the behavior of Information Systems (Roland et al , 2011).

Artefacts used in design science may include algorithms, user interfaces, and system design methodologies.

The Design Science IS framework found as quoted in (Hevner et al, 2004) overlays a focus on three inherent research cycles. .i.e.

i. The Relevance Cycle: This bridges the contextual environment of the research project with the design science activities.
ii. The Rigor Cycle: This connects the design science activities with the knowledge base of scientific foundations, experience, and expertise that informs the research project.
iii. The Design Cycle: This iterates between the core activities of building and evaluating the design artifacts and processes of the research. These three cycles must be present and clearly identifiable in a design science research project.

The following below briefly expands on activities the researcher performed in each cycle.

i. The Relevance Cycle: The researcher gathered requirements from the Case study and also reviewed literature on similar issue tracking systems to clearly establish the Problems and Opportunities. The researcher further tested the system with the users to ensure it addresses the problems.
ii. The Design Cycle: Here the researcher built a prototype based on the requirements and then gave it to the users for evaluation.
iii. Rigor Cycle: Researcher reviewed the results from the testing and evaluation as assessed by the experts in issue tracking systems.

3.3 Evolutionary Prototyping software development life cycle model

The methodology used for this research project is evolutionary Prototyping software development life cycle model. Literature on this evolutionary Prototyping model and why it has been chosen can be found under section 2.7 of this research proposal. Below is an illustration of the development methodology for this project (Breu et al, 2010).

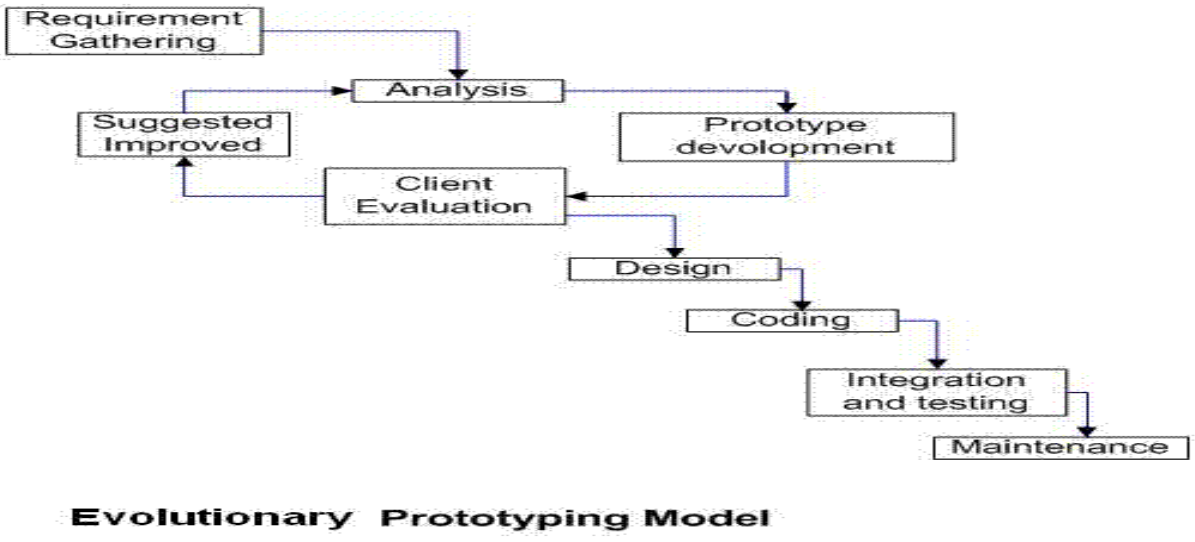

Figure 3.1: Methodology (Evolutionary Prototyping Methodology (Breu et al, 2010).)

3.3.1 Systems Requirements Gathering and Analysis

The main objective of this stage is to document requirements for the software issue tracking system. For the researcher to study and analyze software issue logging and resolution process flow for both existing and needed functionality, the following are methods used:

i. Review of literature
ii. Interviews

To gather complete and solid facts during this phase of requirement gathering and analysis, the researcher ensured that the following are done before and during each method used to gather the requirements:

a. Getting user cooperation
b. Choosing the facts to gather
c. Initiating fact gathering. Through announcements and kick-off meetings by the management.
d. Choosing where to get the facts
e. Introduction to the employees at the case study.
f. Proper recording technique of the facts gathered.
g. Keeping the data collected very well organized.

Review literature

This involved reviewing published literature namely journals and books in regard to software issue tracking and existing software issue tracking system. Literature review helped the researcher to gain a full understanding of the research domain, appreciate what has been done and identify the gaps within the existing systems and hence identify possible improvement options to the subject area. The researcher also got a better understanding of what areas he needs to focus while gathering his interview question for the process.

Specifically the following literature was considered:

a. Previously done works in this area of software issue tracking and software issue tracking system.
b. Methods others have used to tackle similar software issue tracking problems.
c. Software issue tracking in software powered companies.

Interviews

The researcher interviewed core banking system support staff and IT managers at Centenary Bank. This method was chosen because interviews of individuals and small groups require less stakeholder commitment than large workshops. And at the same time, interviews provide an opportunity to explore or clarify topics in more detail. Semi-structured interviews were conducted. The semi-structured interviews were also undertaken in a question-and-answer format. This is more flexible and was rightly used to gather general information about the system (JIRA) currently used at the place of case study. Here the respondents were free to answer in their own words. In this way their views are not restricted. So the researcher got a bigger area to further explore the issues pertaining to a problem.

The researcher also took some time to ask questions at Neptune, the Software vendor for Equinox Core Banking System. While here, the researcher interviewed the Head of Technical support for clarity on how they assign, track, manage and resolve the issues that will have been logged onto JIRA.

Interviewees were selected from the core banking system support staff and IT technical managers and interview was done in one phase that combined gathering and validating of the facts;

A sample interview Guide used by the researcher was attached in the appendixes section.
The context of questions that was used in the interview where circling around practices of issue tracking carried out at the bank.

Documentation of the software issue tracking system requirements and prototype development

After data has been collected and analyzed, the requirements document was drafted, reviewed with software issue tracking system users and research project supervisor. A prototype was developed based on the requirements document and shared the users to certify whether all the requirement had been captured. This prototype was continuously enhanced basing on the feedback of the users until it was able to capture all the necessary requirements

3.3.2 System Design

The purpose of the design phase is to plan a final solution of the problem as was specified by the requirements document. This phase was a main step in modelling out a quality software issue tracking system, and has a major impact on the later phases, particularly implementation, testing and maintenance. The output of this phase was a design document containing a blue print or plan for the solution, and is used later during implementation, testing and maintenance.
The design document included three main architecture models for the software issue tracking system i.e. information model, functional model and data model.

i. The information model contained a high level context diagram showing the software issue tracking system with its possible interactions and functionalities/modules. UML language was used to model the context diagram.
ii. The functional model has a detailed design for the internal logic of each of the modules specified in context model. During this phase further details of the data structures and algorithmic design of each of the modules is specified. UML language was used to model the modules.
iii. The data model was also designed using UML language. This model specifies all the data aspects of the system such as entities involved, relationships between entities and attributes of the entities. This model is inform of a Data Flow Diagram and an Entity Relationship Diagram.

The design models were tested and reviewed with the students and research project supervisor before being passed for the next stage i.e. Implementation / Coding stage.

System Implementation / Coding

Once the design is complete, most of the major decisions about the system would have been made and what is left for this stage is to translate the designs of the system into code and scripts in a given programming language to implement the design in the best possible manner. Since a well written code reduces the testing and maintenance effort, and the testing and maintenance cost of software are much higher than the coding cost, the goal of this phase was to reduce the testing and maintenance effort. Hence, during phase, the focus was on developing programs that are easy to write, simple, clear, and documented.

The researcher implemented the system designs using an MVC (Model View Controller) model. MVC separates the Model (Data), View (Screen/View) and Controller parts of a system making it easy to edit one without affecting the other hence easing maintenance, troubleshooting, and integration.

a) The View which is the user screens/views were implemented using Java Server Pages (JSP).
b) The Controller was implemented using Java Beans.
c) The Model was implemented using MySQL database management system server.

System Integration / Testing

Testing is the major quality control measure that was employed during software development and its basic function was to detect errors in the software. This normally goes hand in hand with integration as at this step, we are aligning and integrating the key processes and functions of the issue tracking system to those of the company/ area where tests was conducted.

During requirement analysis and design, the output is a document that is textual and non-executable. After the implementation, computer programs are available that can be executed for testing phases. This implies that testing was not only used to uncover errors introduced during

coding, but also errors introduced during the previous phases. Thus, the goal of testing was to uncover requirement, design or coding errors in the project. The researcher employed different levels of testing.

The starting point of testing was unit testing. Here, tests were separately performed simultaneously with the coding of each component. After this, the components are gradually integrated into subsystem, which are then integrated and eventually form the entire system. During integration of components, integration testing was performed. The goal of this testing was to detect design errors, while focusing on testing the interconnection between components and also the entire process flow of the system customised for this test area / company.

After the system is put together, general system testing was performed. Here the system was tested against technical system requirements to see if all the requirements are met and the system performs as specified by the requirements document. Finally, acceptance testing is performed to demonstrate to the users and IT managers, in the real life test scenarios with live data.

For testing to be successful, proper selection of test cases is essential. There are two different approaches taken to selecting test cases:

i. Functional (black box) testing. In functional testing the software for the module to be tested is treated as black box, and then test cases are decided based on the specifications of the system or module. The focus is on testing the external behaviour of the system. Functional testing was used for higher levels of testing.
ii. Structural (white box) testing. In structural testing the test cases are decided based on the logic of the module to be tested. Structural testing was used for lower levels of testing.

CHAPTER FOUR

SYSTEM ANALYSIS AND DESIGN

4.0 Introduction

This chapter summarizes the findings of this research. It lists and describes the functions required to improve the issue tracking processes, ensure efficiency and effectiveness in issue tracking systems basing on the requirements gathered. These requirements were gathered by interviews, and literature review as indicated in the methodology chapter (Chapter 3). This section also includes an analysis of the data collected from the case study through interviews and observations that led to the conclusion of the functional and non-functional requirements. This chapter also presents the functional designs, logical database design, entity relationship diagrams, physical database design and user interface design

4.1 Requirements Gathering

Here, the researcher used the two methods as mentioned in the methodology chapter to gather requirements. Each method played a role as indicated below;

4.1.1 Review of literature

This involved reviewing published literature namely journals and books in regard to software issue tracking and existing software issue tracking system. Literature review helped the researcher to gain a full understanding of the research domain, appreciate what has been done and identify the gaps within the existing systems and hence identify possible improvement options to the subject area. But most important of all, the researcher also got a better understanding of what areas he needed to focus on while gathering his interview question for the process.

4.1.2 Interviews

The researcher mainly interviewed core banking system support staff and IT managers at Centenary Bank. Interviews were made using the Interview guide indicated under the methodology chapter above.

The researcher also took some time to interview the Head of Technical support at Neptune (Equinox Core Banking System) for clarity on how they assign, track, manage and resolve the issues that will have been logged onto JIRA.

The table below shows the questions the researcher focused on to help gather requirements;

Interviewer Questioning Context	**Summary Findings From Interviewee's Responses**
What issue tracking system do you use? Have you always used this system in your current position, or have you switched recently?	Issue tracking system being used is called JIRA and they have used it for about 8 years now, in regards to the support of their Core Banking System.
Please give me a tour of the issue tracking system you use	A tour through JIRA brought out that the following issue tracking functionalities were fully supported; Timely Updates, Graphical Display, Reporting, Search Facility.
Can you please describe the primary purpose of issue tracking systems from your perspective?	In their own understanding of issue tracking systems, the various interviewee indicated that an issue tracking system is a system where issues with their Core Banking System are communicated to the system vendor-Neptune, for addressing.
What are the different roles and responsibilities of the people working with the issue tracking system in your department? Do certain people have different levels of access to the issue tracking database than others?	Interviewee from the bank side indicated that business system support staff in IT department, are responsible for logging the issue from the users, interacting with developers, tracking progress of the issue and applying the solution once process is done. Different business system support staff have different levels of access. Only registered users on

a) Is there a QA (Quality Assurance) process? b) Do outside stakeholders (users) have the ability to enter bugs and/or feature requests directly/indirectly into the issue tracking system? c) Does management make use of any reports or metrics from the issue tracking system?	JIRA can enter bugs / requests. Their management uses the reports but not a very large scale. QA on the bank side is done by testing solution before deployment on live environment. Interviewee from the Software vendor side indicated that system support staff in Support department, are responsible for viewing logged issues, interacting with issue loggers, and providing a solution. Different system support staff have different levels of access. Only registered users on JIRA can view and comment on bugs / requests. Their management uses the reports on a very large scale. QA on the software vendor side is done by testing solution before sending to client.
What information is usually tracked with each issue?	Information tracked is; What is the issue, when does it occur, where does it occur, how does it occur, what is the impact of occurrence, who has logged it, who has It been assigned too.
Please walk me through the process of: a) Logging a new issue into the issue tracking system. b) Starting to work on resolving a bug. c) Starting to work on implementing a new feature	On the bank side, It was clear that process of logging a new issue was well understood by the majority however they were not good at attaching the relevant information. Also when it came to implementing a solution, it was better when the developer did this themselves
What is involved in the process of assigning a bug to a user and what is taken into consideration when choosing a resolver to handle the bug?	On the bank side, the business system support staff simply logs the issue on JIRA and assign it to the Software Vendor-Neptune.

a) Experience on similar types of issues? b) Amount of work they are currently handling? c) Others please specify is available.	On the Software Vendor side-Neptune, the Moderator/Head of Support manually assigns the issue basing on: Area of expertise, Experience on similar types of issues, Amount of work they are currently handling, who they feel can deliver on time, intuition, etc.This is not done automatically.
What communication channels do you use to communicate with your colleagues and other project stakeholders? How frequently do you use each communication channel?	Common Channels used for communication between the bank's business system support staff and Software vendors is Email and Instant messaging such as Skype, verbal conversation - (face-to-face, phone/Skype)
Do you have a preference as to which communication channel you use? If so, under what conditions and why? a.	The preferred communication channel is Email simply because Emailing is a formal and professional way of exchanging important communication is the business world, Plus it provides an archived trail / communication history for future references.
When reporting an issue, how does the system handle duplicate reporting?	When an issue is reported, regardless of whether it has been logged before or resolved before, JIRA accepts the issue and assigns it to the resolver who may later learn that it was resolved before, or even never get to know it was resolved before meaning he goes through the trouble shooting and resolution process again till a solution is provided.
What types of information do you frequently require from others in order to complete your day-to-day issue tracking and related software development tasks? How do you usually obtain this	Information that Neptune Staff (Software vendor) often require includes: instructions on how to reproduce the issue, the severity of the bug, Corresponding Screenshot/Attachment/logs can also be uploaded capturing the actual

information? How frequently do you find yourself looking for these kinds of information	operation/output/message. This information is key to any issue resolution.

Table 4.1: Results from the interview

Below is a summary of the features identified from the interviews and interaction with the key identified staff for each process. It's from these results that functional and non-functional requirements were identified.

Tracking Tool	Timely Update	Graphical Display / Reporting	Search Facility	Auto Issue assignment	Duplicate bug filtering	Bug Prediction	Change control
JIRA	Yes	Yes	Yes	No	No	No	Yes
Octopus (Proposed Issue Tracking System)	Yes	Yes	No	Yes	No	Yes	Yes

Table 4.2: Summary of the results in terms of functional requirements

4.2 Functional Requirements

1. There is a function to handle auto-ticket assignment. This will be based on the information supplied and will help quicken the ticket resolution process.
2. The validator function shall also be responsible for reviewing duplicates in tickets logged. This will help with quick issue resolution and data integrity.
3. There is a function for performing bug prediction basing on recently resolved bugs
4. There is a function for tracking change control process of patches being deployed to resolve issues.
5. The system is able to capture necessary information pertaining a ticket on a project. All tickets will be specific to a project.
6. There is a function that will verify information captured and ensure it is correct and valid.
7. There is a function responsible for tracking status of a ticket and alert the responsible users whenever necessary and need arises.
8. There is a function to review any outstanding tickets and report accordingly. If need be, the function will reassign the tickets.
9. There shall be a function that will provide an external API for third party applications to interact with the system.

4.3 Hardware and Software Requirements

The IT personnel were also interviewed as indicated on Appendix 1, 2 and 3 under the section of Hardware and Software availability to ascertain the available hardware and software. A business process workflow management system to automate business processes seamlessly should have the following minimum hardware and software requirements:

The client will contribute to relevant process flows in a manner that helps control operational over heads and risks.

The minimum entry level server hardware and software configuration required to run consolidated services are as follows;

i. Intel(R) Core 2 Duo CPU/1.8Ghz or Higher
ii. 16 GB RAM or Higher
iii. 1000GB hard-disk
iv. Monitor, Keyboard and Mouse

v. Windows 2008 Server or Higher

vi. Network Interface Card(s)

vii. Secure connection links to the remote terminals

4.4 Non Functional Requirements

4.4.1 Authentication model

The system has an authentication mechanism for user verification and validation so as to allow users to access resources based on their system access rights and roles. Although this authentication model primarily helps prevent unauthorized system access, it further kick starts the issue logging process by channeling the user to their respective screen and pushing their input to the relevant process queues as will be demonstrated.

Below is a representation of the functional design model.

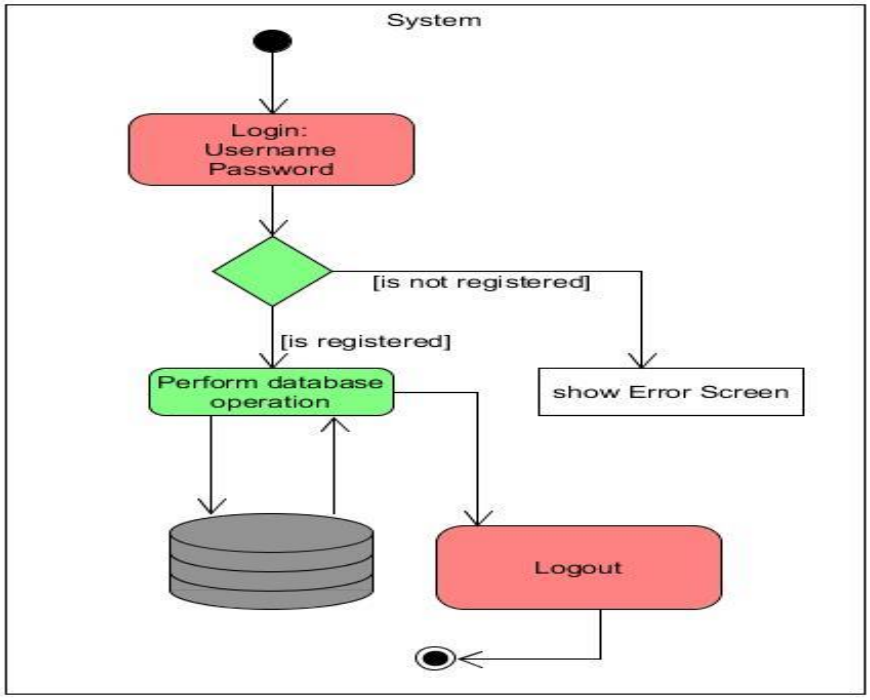

Figure 4.0: authentication model

4.4.2 User Error handling model

The system has a proper error or exception handling process on any malfunctions so as to alert the user of the exception. Octopus supports three types of error handling modes,

a) Logging errors on a log file that can be shared with the system maintainer.
b) Displaying alert dialogs on the user interface.
c) Sending email notifications to the maintainer in cases of critical system processing errors.

The figure below shows how the system shall handle error and exceptions encountered during logic processing.

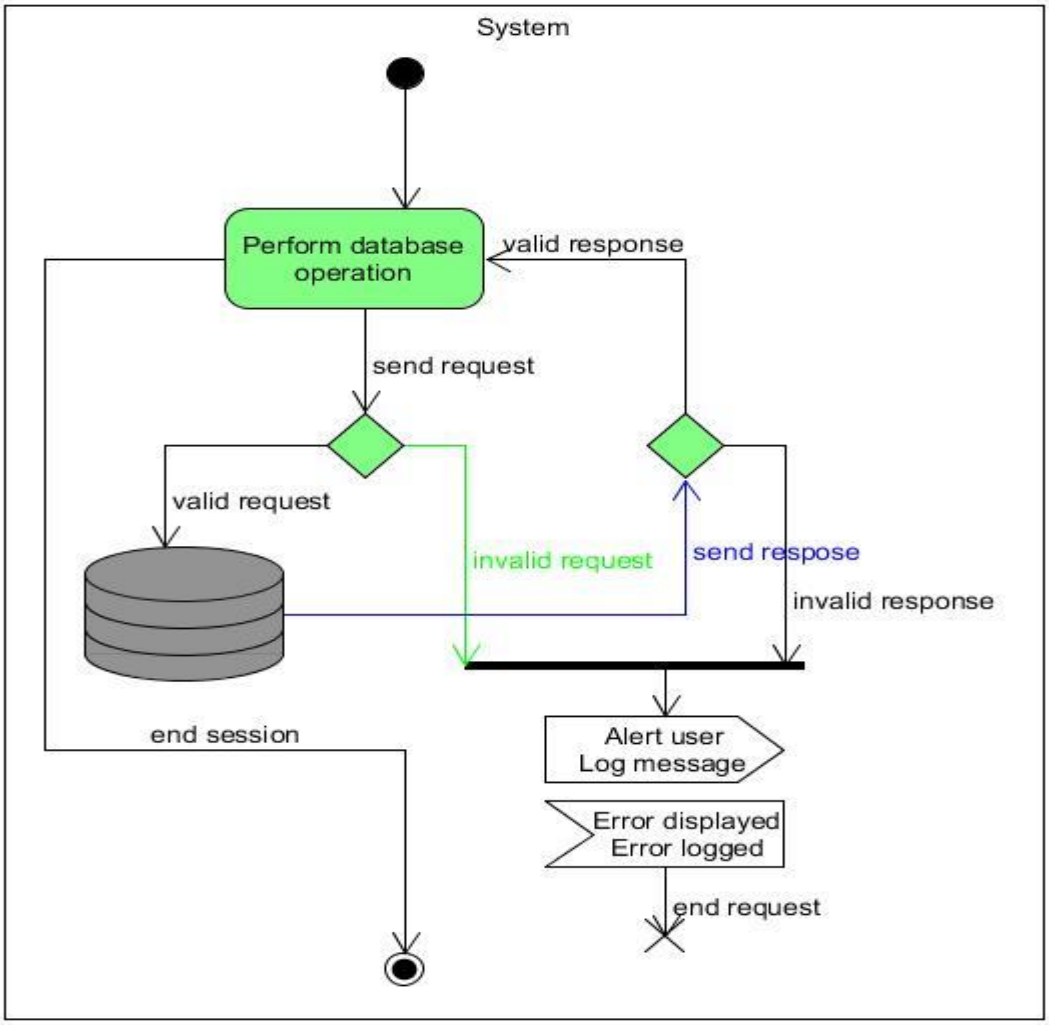

Figure 4: Error handling model

4.4.3 Validation of user roles

The system also has the ability to validate and manage user roles and system access depending on the user role assigned by the system admin.

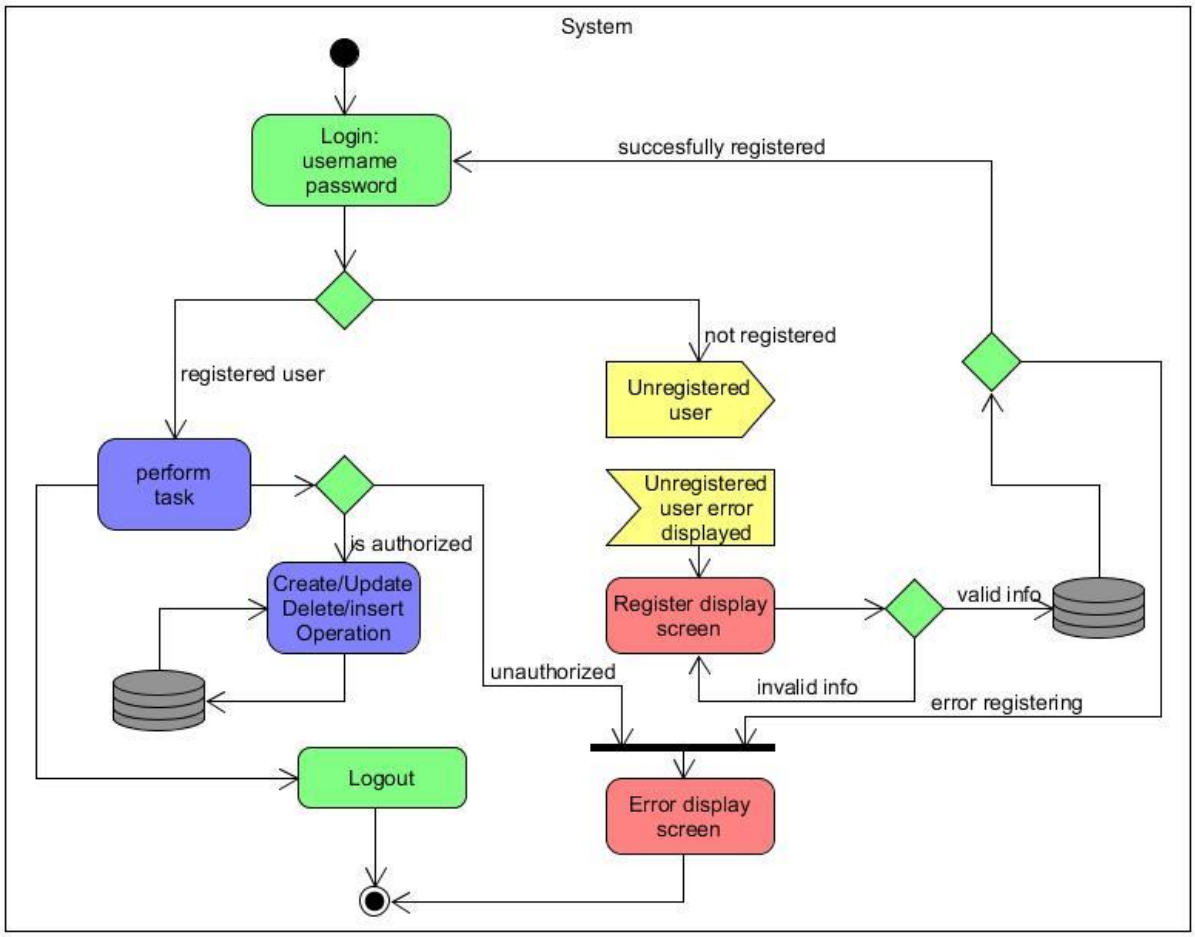

Figure 4.2: validation model

This functionality allows for the system to validate and process valid and authorized user requests.

4.5 Functional Design

This design portrays the different processes that will allow the user to interact with the Octopus issue tracking system.

This was illustrated using a context diagram i.e. context level 0 and context level 1. The overall user and system interaction is demonstrated below. This involves an active authenticated user initiating the process of bug tracking.

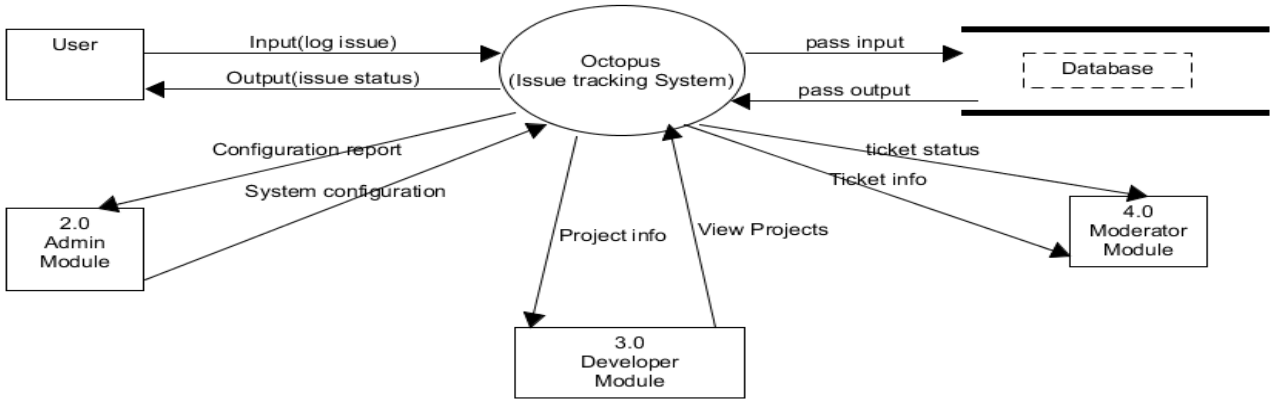

Figure 4.3: Context Diagram (Level 0)

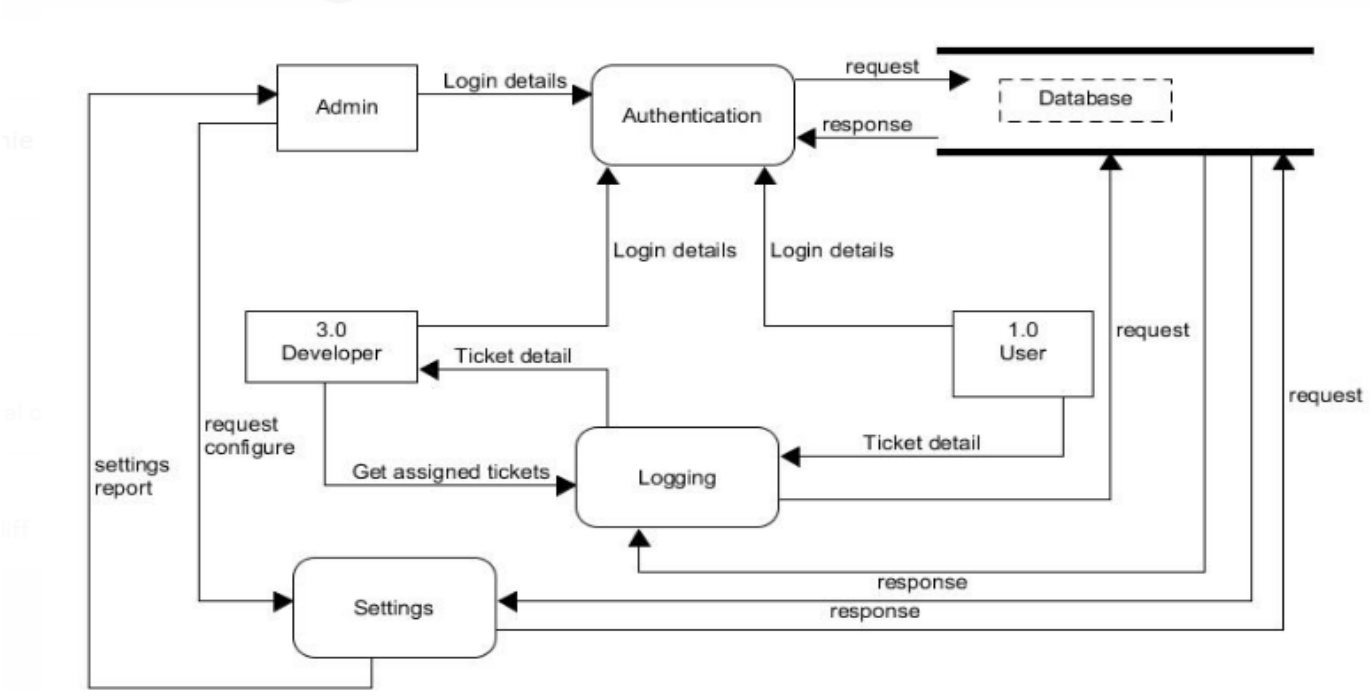

Figure 4.4: Context Diagram, Level 1(Data Flow Diagram)

4.5.1 System Functional Design

The following system functionalities is be availed;

1. There is a function to handle auto-ticket assignment. This will be based on the information supplied and will help quicken the ticket resolution process.
2. The validator function shall also be responsible for reviewing duplicates in tickets logged. This will help with quick issue resolution and data integrity.
3. There is a function for performing bug prediction basing on recently resolved bugs
4. There is a function for tracking change control process of patches being deployed to resolve issues.
5. The system is able to capture necessary information pertaining a ticket on a project. All tickets will be specific to a project.
6. There is a function that will verify information captured and ensure it is correct and valid.
7. There is a function responsible for tracking status of a ticket and alert the responsible users whenever necessary and need arises.
8. There is a function to review any outstanding tickets and report accordingly. If need be, the function will reassign the tickets.
9. There shall be a function that will provide an external API for third party applications to interact with the system.

4.5.2 Database Design

4.5.2.1 Logical Database Design

This subsection of the design shows the attributes of every entity in the database. Primary keys are bolded and foreign keys are underlined. The system under study will be referred to as Octopus and its logical model is described below.

a) This table below captures the details of a user of the system.

SYSUSER {**Id**, CODE, PASSWORD, EMAIL, CREATE_DT, IS_ADMIN, ACCESS_LEVEL, STATUS, FIRST_NM, LAST_NM, GENDER, PHONE, LAST_MOD_DT, SUPERVISOR_ID }

b) This table captures information regarding a project created by a system user.

Project {**Id**, NAME, PROJ_DESC, CREATE_DT, LAST_MOD_DT, STATUS, OWNER, VISIBILITY, ACCESS, PARTICIPANTS, URL, CREATOR, ARCHITECTURE}

c) This table captures information regarding a user access defined by a system owner/admin.

SysAccess {**Id**, LEVEL, LEVEL_DESC, STATUS, MAX_ENTRIES, CREATE_DT, LAST_MOD_DT, LEVEL_SHORT_NM}

d) This table captures information regarding a ticket created on a project by a system user

Ticket {**Id**, NAME, TICKET_DESC, PROJECT_ID, PROJ_CONTACT, PRIORITY, CATEGORY, CREATOR, OWNER, STATUS, RESOLVE_DT, LOG_DATE, LAST_MOD_DT, TARGET_DT, LOG_FILE, DOC_FILE}

e) This table captures information regarding a comment on a ticket created on a project by a system user

Comment {**Id**, COMMENT_NM, COMMENT_DESC, COMMENTER_REF, TICKET_REF, CREATE_DT, ATTACHMNT, STATUS}

f) This table captures the patches delivered on a ticket by a resolver.

TICKET_PATCH {**Id**, NAME, HASH_REF, PATCH_LOC, OWNER_ID, LAST_MOD_DT, TICKET_ID}

g) This table captures all system alerts, logs and notifications; it's also used to place system and user mail requests to be forwarded

Alerts {**Id**, SUBJECT, MESSAGE, RECIPIENT, STATUS, LOG_DT, LAST_MOD_DT, ATTACHEMENT}

h) This table captures all access levels defined for the various user groups.

Alerts {**Id**, USR_ID, LEVEL_ID, STATUS, CREATE_DT, LAST_MOD_DT, LAST_MOD_DT, ATTACHEMENT}

i) This view is used to compute user to ticket statistics

USER_STATS {Id, CODE, EMAIL, TOTAL_TICKETS, OPEN_TICKETS, CLOSED_TICKETS, PERCENT_OPEN, PERCENT_CLOSED}

4.5.2.2 Physical Database Design

Table Name	SQL Script for implementation.
ACCESS	CREATE TABLE `access` (`ID` int(11) NOT NULL AUTO_INCREMENT, `USR_ID` int(11) NOT NULL, `LEVEL_ID` int(11) NOT NULL, `STATUS` varchar(10) NOT NULL, `CREATE_DT` date NOT NULL, `LAST_MOD_DT` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP, PRIMARY KEY (`ID`), UNIQUE KEY `USR_ID` (`USR_ID`)) ENGINE=InnoDB DEFAULT CHARSET=latin1
ALERTS	CREATE TABLE `alerts` (`ID` int(11) NOT NULL AUTO_INCREMENT, `SUBJECT` varchar(50) NOT NULL, `MESSAGE` text NOT NULL, `RECIPIENT` text, `STATUS` char(1) NOT NULL, `LOG_DT` date NOT NULL, `LAST_MOD_DT` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP, `ATTACHEMENT` text, PRIMARY KEY (`ID`)) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=latin1
COMMENT	CREATE TABLE `comment` (`ID` int(11) NOT NULL AUTO_INCREMENT,

	`COMMENT_NM` varchar(300) NOT NULL, `COMMENT_DESC` text NOT NULL, `COMMENTER_REF` int(11) NOT NULL, `TICKET_REF` int(11) NOT NULL, `CREATE_DT` date NOT NULL, `STATUS` char(1) NOT NULL DEFAULT 'N', `ATTACHMNT` varchar(200) DEFAULT NULL, PRIMARY KEY (`ID`)) ENGINE=InnoDB DEFAULT CHARSET=latin1
PROJECT	CREATE TABLE `project` (`ID` int(11) NOT NULL AUTO_INCREMENT, `NAME` varchar(150) NOT NULL, `PROJ_DESC` text, `CREATE_DT` date NOT NULL, `LAST_MOD_DT` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP, `STATUS` enum('active','closed','watching') NOT NULL, `OWNER` int(11) NOT NULL, `VISIBILITY` enum('private','public') NOT NULL, `ACCESS` enum('read','write','admin') NOT NULL, `participants` text, `url` varchar(150) DEFAULT NULL, `creator` int(11) NOT NULL, `ARCHITECTURE` text, PRIMARY KEY (`ID`), UNIQUE KEY `NAME` (`NAME`)) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=latin1
SYSCCESS	CREATE TABLE `sysaccess` (`ID` int(11) NOT NULL AUTO_INCREMENT,

	`level` int(11) DEFAULT NULL, `LEVEL_DESC` text, `STATUS` varchar(10) NOT NULL, `MAX_ENTRIES` int(11) NOT NULL, `CREATE_DT` date NOT NULL, `LAST_MOD_DT` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP, `LEVEL_SHORT_NM` varchar(150) DEFAULT NULL, PRIMARY KEY (`ID`), UNIQUE KEY `LEVEL` (`level`), UNIQUE KEY `level_2` (`level`), FULLTEXT KEY `LEVEL_SHORT_NM` (`LEVEL_SHORT_NM`), FULLTEXT KEY `LEVEL_DESC` (`LEVEL_DESC`)) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=latin1
SYSUSER	CREATE TABLE `sysuser` (`ID` int(11) NOT NULL AUTO_INCREMENT, `CODE` varchar(10) NOT NULL, `PASSCODE` varchar(200) NOT NULL, `EMAIL` varchar(150) NOT NULL, `CREATE_DT` date NOT NULL, `IS_ADMIN` int(11) NOT NULL DEFAULT '1', `ACCESS_LEVEL` int(11) NOT NULL, `STATUS` varchar(10) NOT NULL, `FIRST_NM` varchar(25) DEFAULT NULL, `LAST_NM` varchar(25) DEFAULT NULL, `GENDER` char(1) DEFAULT NULL, `PHONE` varchar(15) DEFAULT NULL,

	`LAST_MOD_DT` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP, `SUPERVISOR_ID` int(11) DEFAULT NULL, PRIMARY KEY (`ID`), UNIQUE KEY `CODE` (`CODE`), UNIQUE KEY `EMAIL` (`EMAIL`)) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=latin1
TICKET	CREATE TABLE `ticket` (`ID` int(11) NOT NULL AUTO_INCREMENT, `NAME` varchar(300) DEFAULT NULL, `TICKET_DESC` text NOT NULL, `PROJECT_ID` int(11) NOT NULL, `PROJ_CONTACT` varchar(50) DEFAULT NULL, `PRIORITY` enum('critical','high','moderate','low') NOT NULL, `CATEGORY` enum('project','knowledge gap','enhancement','bug','other') DEFAULT NULL, `CREATOR` int(11) NOT NULL, `OWNER` int(11) NOT NULL, `STATUS` enum('open','closed') DEFAULT NULL, `RESOLVE_DT` date DEFAULT NULL, `LOG_DATE` datetime DEFAULT NULL, `LAST_MOD_DT` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP, `TARGET_DT` date DEFAULT NULL, `LOG_FILE` varchar(50) DEFAULT NULL, `DOC_FILE` varchar(50) DEFAULT NULL, PRIMARY KEY (`ID`), FULLTEXT KEY `NAME_3` (`NAME`,`TICKET_DESC`), FULLTEXT KEY `TICKET_DESC` (`TICKET_DESC`)

	) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=latin1
TICKET_PATCH	CREATE TABLE `ticket_patch` (`ID` int(11) NOT NULL AUTO_INCREMENT, `NAME` varchar(30) NOT NULL, `HASH_REF` text, `PATCH_LOC` text, `OWNER_ID` int(11) NOT NULL, `LAST_MOD_DT` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP, `TICKET_ID` int(11) NOT NULL, PRIMARY KEY (`ID`)) ENGINE=InnoDB DEFAULT CHARSET=latin1
USER_STATS	CREATE ALGORITHM=UNDEFINED DEFINER=`root`@`localhost` SQL SECURITY DEFINER VIEW `user_stats` AS select distinct `usr`.`ID` AS `ID`,`usr`.`CODE` AS `CODE`,`usr`.`EMAIL` AS `EMAIL`,(select count(0) from `ticket` where (`ticket`.`OWNER` = `usr`.`ID`)) AS `TOTAL_TICKETS`,(select count(0) from `ticket` where ((`ticket`.`OWNER` = `usr`.`ID`) and (`ticket`.`STATUS` = 'open'))) AS `OPEN_TICKETS`,(select count(0) from `ticket` where ((`ticket`.`OWNER` = `usr`.`ID`) and (`ticket`.`STATUS` = 'closed'))) AS `CLOSED_TICKETS`,(select round(((count(0) * 100) / (select count(0) from `ticket` where (`ticket`.`OWNER` = `usr`.`ID`) having (count(0) > 0))),2) from `ticket` where ((`ticket`.`OWNER` = `usr`.`ID`) and (`ticket`.`STATUS` = 'open'))) AS `PERCENT_OPEN`,(select round(((count(0) * 100) / (select count(0) from `ticket` where (`ticket`.`OWNER` = `usr`.`ID`) having (count(0) > 0))),2) from `ticket` where ((`ticket`.`OWNER` = `usr`.`ID`) and

	(`ticket`.`STATUS` = 'closed'))) AS `PERCENT_CLOSED` from `sysuser` `usr` group by `usr`.`ID`
Table Name	SQL Script for implementation.
ACCESS	CREATE TABLE `access` (`ID` int(11) NOT NULL AUTO_INCREMENT, `USR_ID` int(11) NOT NULL, `LEVEL_ID` int(11) NOT NULL, `STATUS` varchar(10) NOT NULL, `CREATE_DT` date NOT NULL, `LAST_MOD_DT` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP, PRIMARY KEY (`ID`), UNIQUE KEY `USR_ID` (`USR_ID`)) ENGINE=InnoDB DEFAULT CHARSET=latin1

Table 4.3: Physical Database Design

4.5.2.3 Entity Relationship Diagram

The Entity Relationship Diagram, often an extension of the logical data model is a graphical representation of the entities and their relationships with one another within the Octopus issue tracking system. It helps in the organization of data in the database. The Octopus issue tracking system will have a database called Octopus and in it will be at least 18 entities that will relate to one another as shown in the Entity Relationship Diagram below.

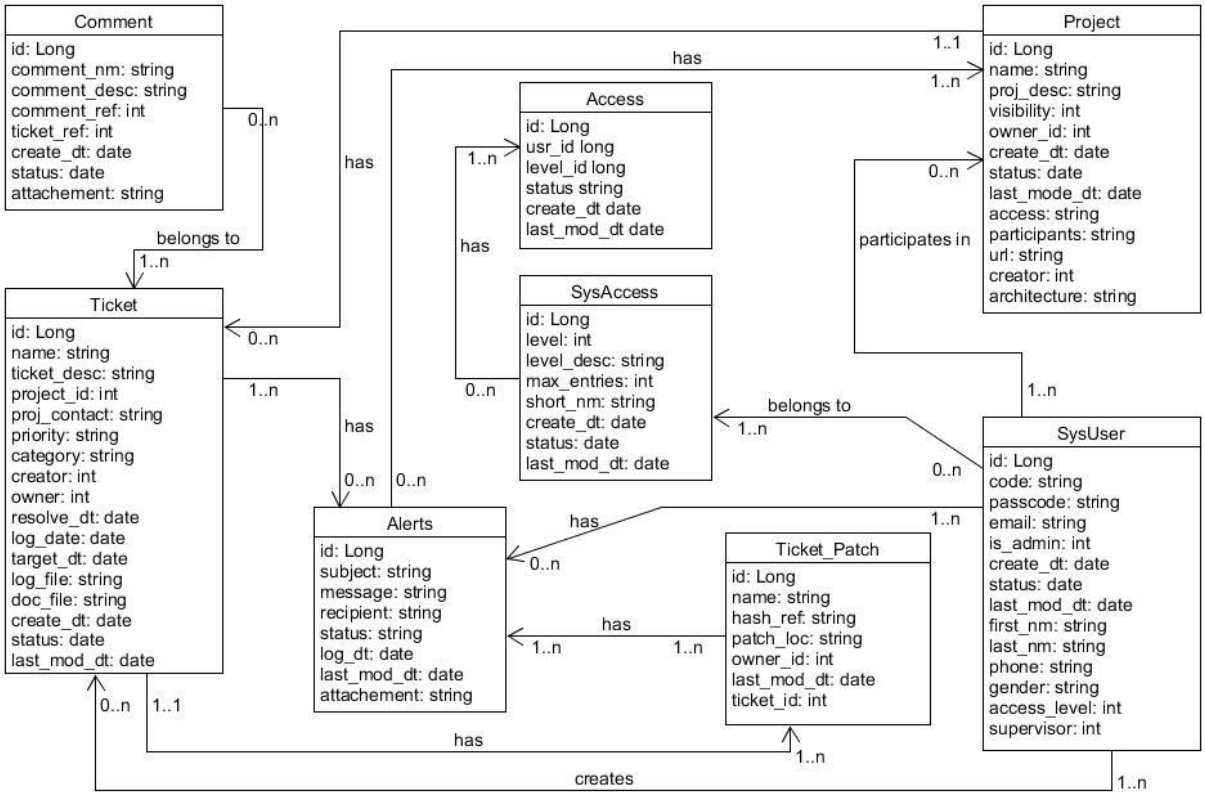

Figure 4.5: Entity Relationship Diagram

4.6 User Interface Design

The user interfaces are intended to help the user access the system in an organized and friendly way. It provides a user friendly format and display of system components and information with which a user can interact with the system components and database in order to provide and process the required services.

4.6.1 Welcome/Login Page

The welcome page will be the home and login page as well for the Octopus issue tracking system. This page also has link to register a user, Projects which allows user to view all public project listings.

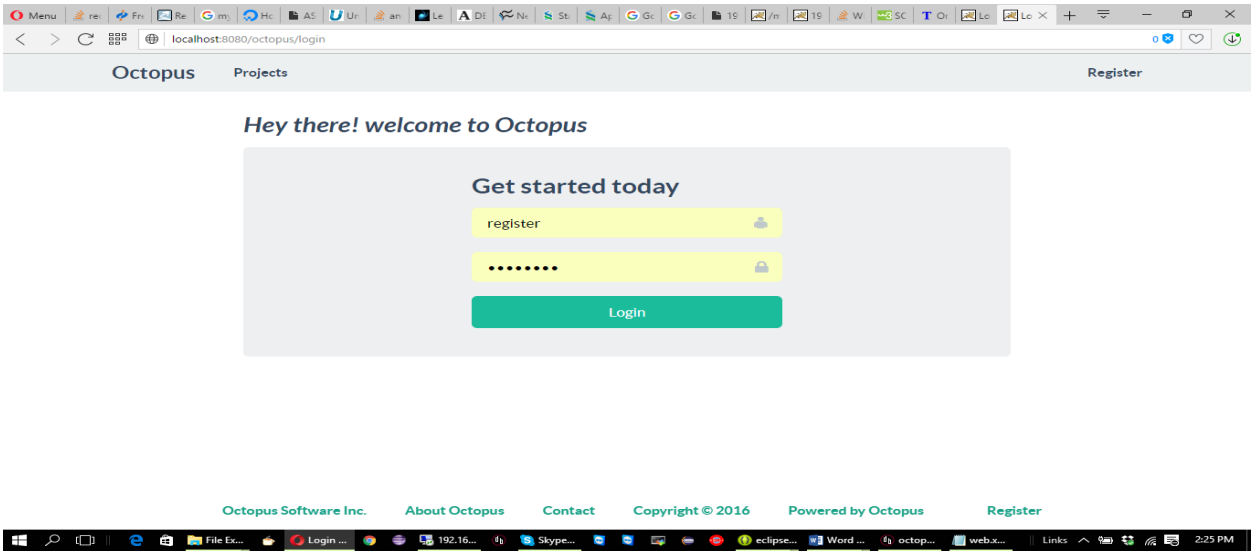

Figure 4.6: Welcome/Login Page

4.6.2 Dashboard Page

This page will show a summary of all the projects, tickets and ticket progress in a general overview for the logged in user.

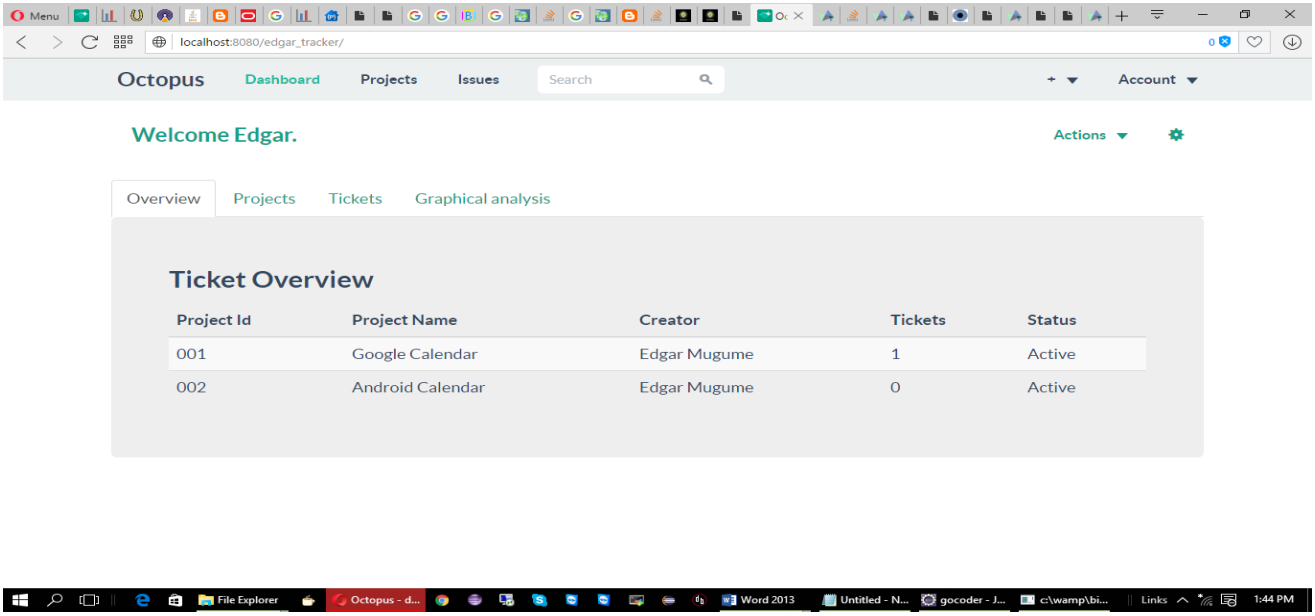

Figure 4.7: Dashboard Page

4.7 Network and System Design

System architecture defines the technical environment of the system and in cooperates the network connection model, hardware and software specifications, the design tools used and the security design model. The diagram below is an illustration of the components and how they fit together. The network model describes the communication infrastructure layout on which the new system is to operate. The components of the model include but not limited to:

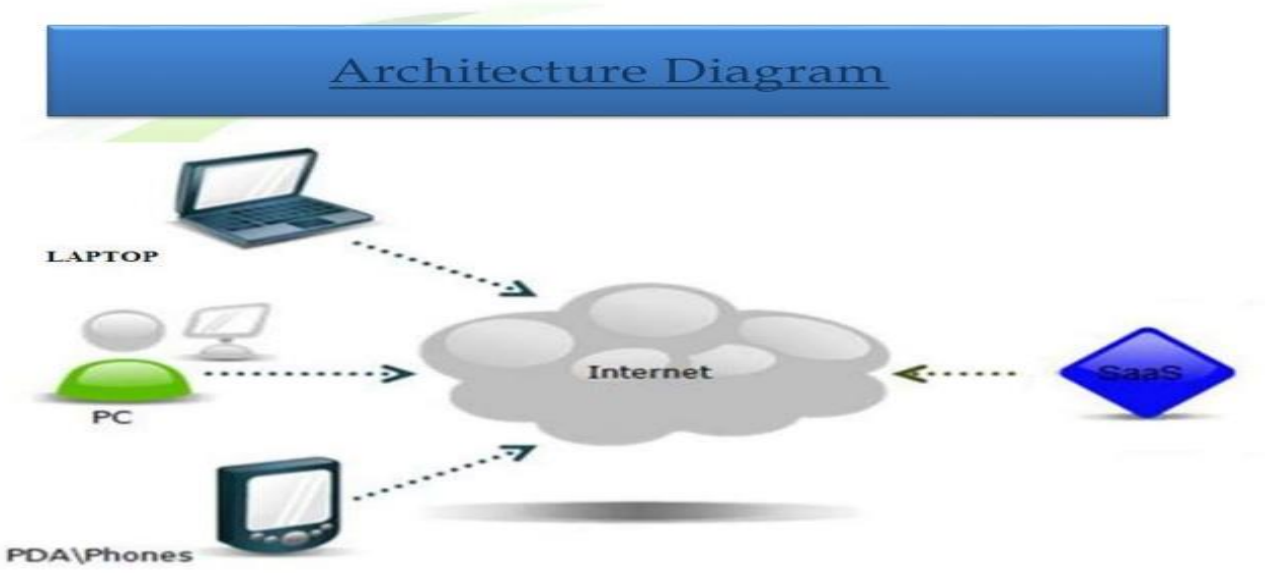

Figure 4.8: Network and System Design

System users or clients and other external stakeholders who have access to computers with an internet connection can access the Octopus issue tracking web application and service from anywhere. The system admin and other users such as reporters, resolvers or developers can also access the system from both within the organization intranet and the outside using the internet. They will use either their personal computers or company computers to access the platform. These computers come in two main forms, laptops which are light weight and can be carried around or desktop computers which are stationary running either Windows, Linux or MacOSx.

A network router will provide the network service access to the client machines on the network preferable a CISCO router LAN.

4.8 System Security

In every data related system, security is a major non-functional requirement especially when lots of people can access the system and run tests. Hackers and other network sniffers can particularly be a threat to the system hence requiring proper security measures to be put in place. Among these security features is data encryption, firewalls and other data verification processes and sub routines involved in securing data.

Octopus applies security measures at the physical and logical levels. The physical level involves lock and restrain of physical Octopus servers from general access. This will only permit authorized personnel to access the servers and make changes.

Logical level of security tightly couples with the network security involved. Securing the network or LAN and adding layers of encryption to the communication protocols and firewalling the network from external unpermitted access will greatly improve the security level of the Octopus issue tracking system.

The other aspect of the Logical security enhancement involves having a proper authentication sub system in Octopus system for session handling and access control into the system for instance, maximum number of failed login attempts, password strengths and combinations.

Finally, there will be a sub routine or service to perform daily backup of the system to ensure that it can be restored should need arise. The process will be automated and will work in two ways. First one being database level replication to other databases on different sites, and the second being dumps of the database and application taken and backed up on different network server. Uninterrupted power supplies will be availed to help sustain the system in case there is a short power outage so the system can go on running as expected.

4.9 Conclusion

This chapter has the researcher providing a description of the logical database and how it will be implemented. It's further culminated into the entity relationship diagram. In order for the users to interact with the system, the researcher proposed the designs of the user interfaces including a report that can accessed by the users. With this guide, the researcher and the system developer should be able to implement the Octopus issue tracking system as described in this chapter.

CHAPTER FIVE

SYSTEM IMPLEMENTATION

5.0 Introduction

This chapter provides a description of the steps that where undertaken to enable the coding of the Octopus issue tracking system and chapter discusses the conversion of the system designs into a proper working issue tracking system. These include the acquisition of the implementation tools and steps for the installation of the acquired development software. In this chapter, the researcher also provides a description of the implementation of the databases and the user interfaces and finally the descriptions of the processes that can be undertaken when installing the system on another computer as expected.

5.1 Acquisition of development tools

Apache tomcat and Eclipse IDE are the software of choice, both being open source, versatile and free to use with a wide market share. Eclipse being the leading IDE for Java development and tomcat, a robust efficient and out of the box servlet engine.

Both these software were acquired from the internet, Eclipse being downloaded from the official Eclipse Software Foundation site and Apache Tomcat 7 from The Apache Software Foundation, both companies renowned for their contributions to the open source community.

Due to time constraints tied to the project life cycle, the researcher sourced support from a systems developer to assist in the development of the system. The development took approximately 2 weeks. The developer involved used the test driven development to quicken the process.

5.2 Installation of the development tools

The researcher installed the development tools mentioned above so as to initiate the development process involving the coding phase and testing as well. Unit testing helped quicken the process where each functionality developed emerged from its corresponding Testcase using the Junit

testing library. The steps below provide instructions on how to install the required development tools priori to development.

5.2.1 The Eclipse Integrated development environment Neon Release (4.6.0)

1. Ensure you have an active internet connection on your computer
2. Open any browser of your choice preferably Chromium based products such as Google Chrome or Opera.
3. Launch Opera, and visit http://eclipse.org and click on the download button to get the latest Eclipse Java EE IDE for Web Developers.
4. Once the download has completed, unzip the folder and double click on the eclipse.exe to launch the environment.
5. Download Java from http://oracle.com if you don't already have it. Ensure to download the right eclipse architecture version for the Java version. The instructions are stated clearly on the eclipse website.

5.2.2 The Apache Tomcat servlet engine Release (7.0.7.0)

1. Once you've installed Eclipse, it's time to install Tomcat from Apache as your servlet engine. Open https://tomcat.apache.org/download-70.cgi to get Tomcat version 7.
2. Download the widows services version for 32/64bit Windows Service installer to a location you can easily access and have write access to.
3. Double click on the installer and install following the default process. The default selections are efficient enough for starters and should get you started.
4. The figure below shows the tomcat installation screens. Check the Host Manager option to enable remote installation of web apps over the web interface.
5. Define a username and password to use to access when deploying web apps to the tomcat servlet engine as in figure 5.3

Figure 5.1 Installation of Tomcat

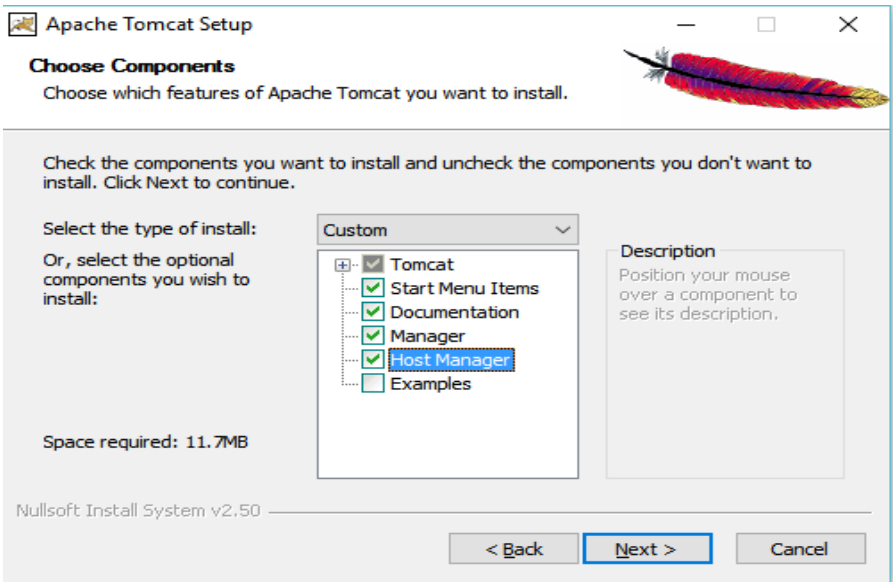

Figure 5.2 Installation of Tomcat

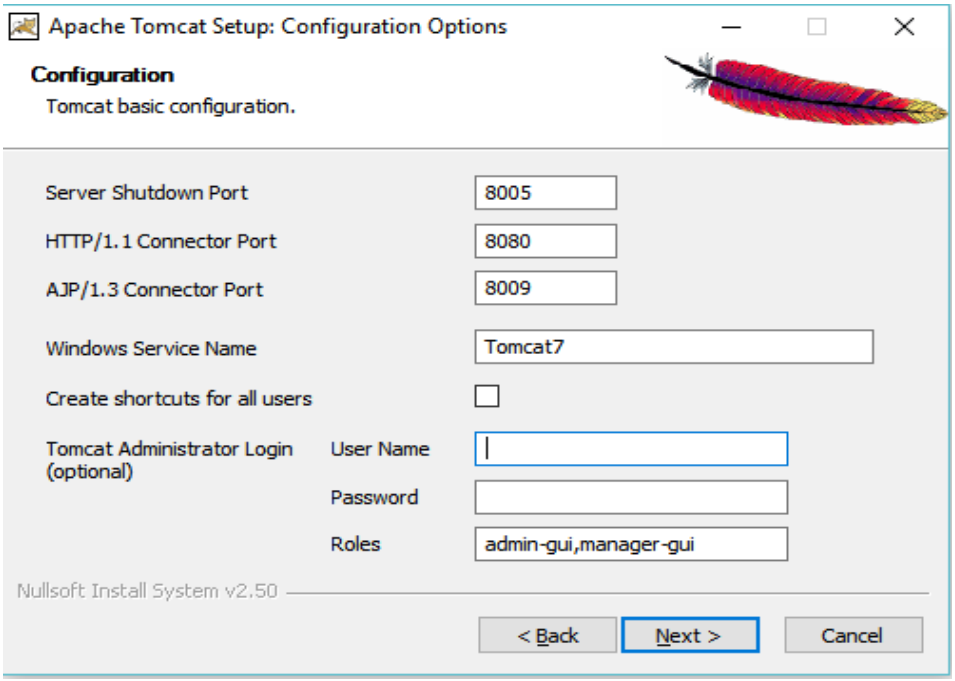

Figure 5.3: Define username/password for tomcat access

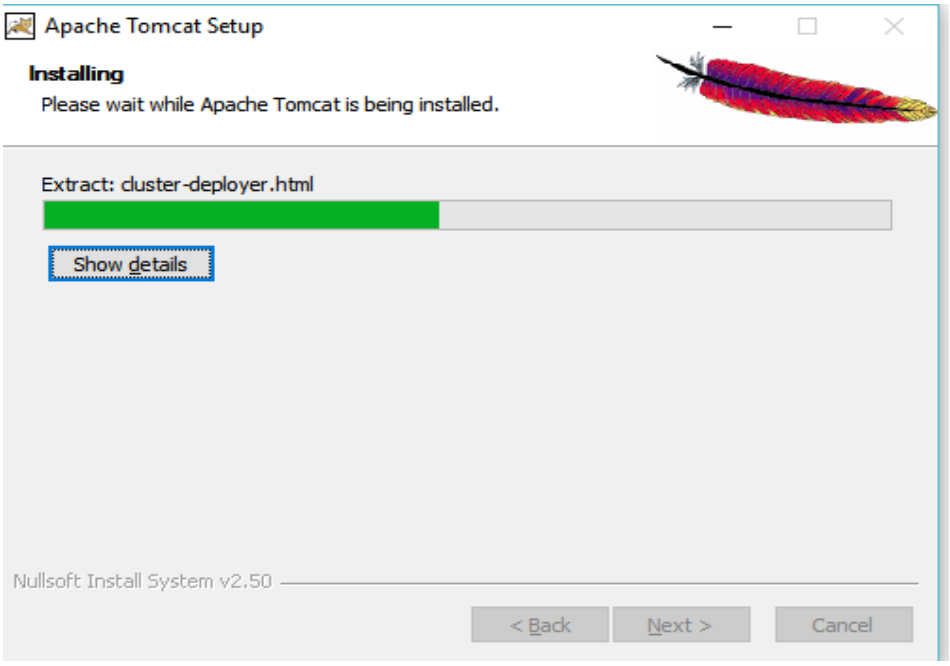

Figure 5.4: Installation process

After the installation is complete, you can find Tomcat under the directory you specified earlier during the installation process from which you can launch the service control using the services.msc command and start or stop tomcat from.

Octopus uses a built in database SQLite but supports other database engines such as MySQL, PostgreSql, Oracle and Sybase out of the box among other. By default Octopus uses the SQLite database engine that comes embedded within the application for a quick start.

5.3 The Coding phase

In this phase, the physical design was structured into functional computer code following the object oriented approach as the languages chosen adhere to the Object oriented programming principal. This allows for reusability and code maintenance among other features. The coding was also driven by unit testing using the Junit testing library. This allowed for development of functionality needed and removed any redundant unused code.

5.3.1 Database Implementation.

The Octopus issue tracking system is partly composed of a database whose name can be changed and specified in the configuration but the default name used is Octopus_db following the application name. This database was primarily designed using the SQLite database but has in-built conversions to handle the different database vendor data types and structure. Supported database vendors include SQLite, Oracle, MySQL, Sybase, PostgreSql and SQLServer. Octopus creates its own tables provided the database type and name are specified. The tables in the database correspond to the Entity classes in the Application and migration is handled automatically by the Octopus SQL engine.

The database has nine entities as shown in figure 4.3 above. This was followed by testing the constraints defined through attempting to enter sample data into random tables with foreign keys and the results were as expected. The data was rejected with foreign key attribute errors thrown. Below is a screen caption of the database tables in the SQL Table generation engine.

```
protected static void setupTables(ConnectionSource connectionPool) {
    try {
        TableUtils.createTableIfNotExists(connectionPool, UserCred.class);

        TableUtils.createTableIfNotExists(connectionPool, Project.class);

        TableUtils.createTableIfNotExists(connectionPool, Ticket.class);

        TableUtils.createTableIfNotExists(connectionPool, Comment.class);

        TableUtils.createTableIfNotExists(connectionPool, Alert.class);

        TableUtils.createTableIfNotExists(connectionPool, UserRole.class);

        TableUtils.createTableIfNotExists(connectionPool, Status.class);

        TableUtils.createTableIfNotExists(connectionPool, Priority.class);

        TableUtils.createTableIfNotExists(connectionPool, ControlParam.class);

    } catch (SQLException e) {
        System.err.println(e.getMessage());
    }
}
```

Figure 5.3 Database Implementation

5.3.2 User interface and logic.

The Octopus system deployment is handled by tomcat web interface. The Host Manager Interface helps with the deployment of web apps to the tomcat servlet engine. Eclipse IDE was used to develop the web app and the SQL engine, Bootstrap and JQuery JavaScript and Cascading Style sheets libraries were used to beautify and validate the entries made. The validation was further managed at the server side using Java and SQL constraints defined. Sample code for the various app files is in the appendix E.

The system is a browser based and as such, a user will need to have a browser installed on their machine. Supported browsers include Firefox, Google Chrome, Chromium, Opera and any other browsers that can support JavaScript. The user can then access the system web portal through the URI relative to the application host machine. IF the instance is running on their machine,

http://localhost:8080/octopus can be used otherwise the Internet protocol or IP of the server hosting the Tomcat instance on which Octopus is deployed has to be specified. Once one has loaded the application URI, they will be presented with the screen as in the figure below.

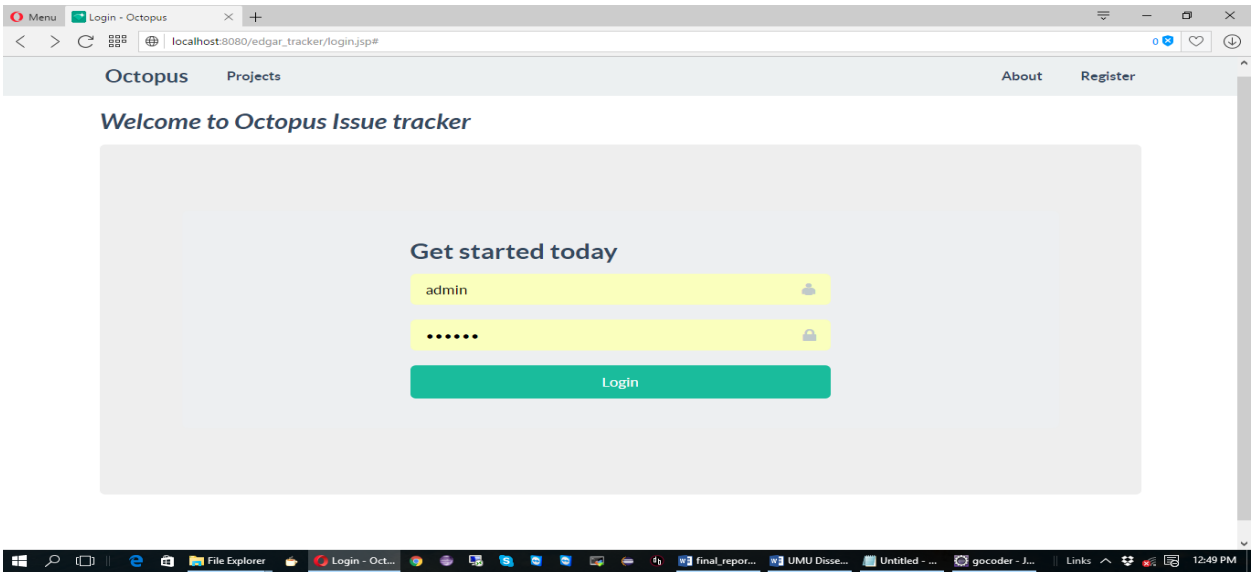

Figure 5.5: Login screen.

On successful login, depending on the user access role, they will then be presented with the screen below.

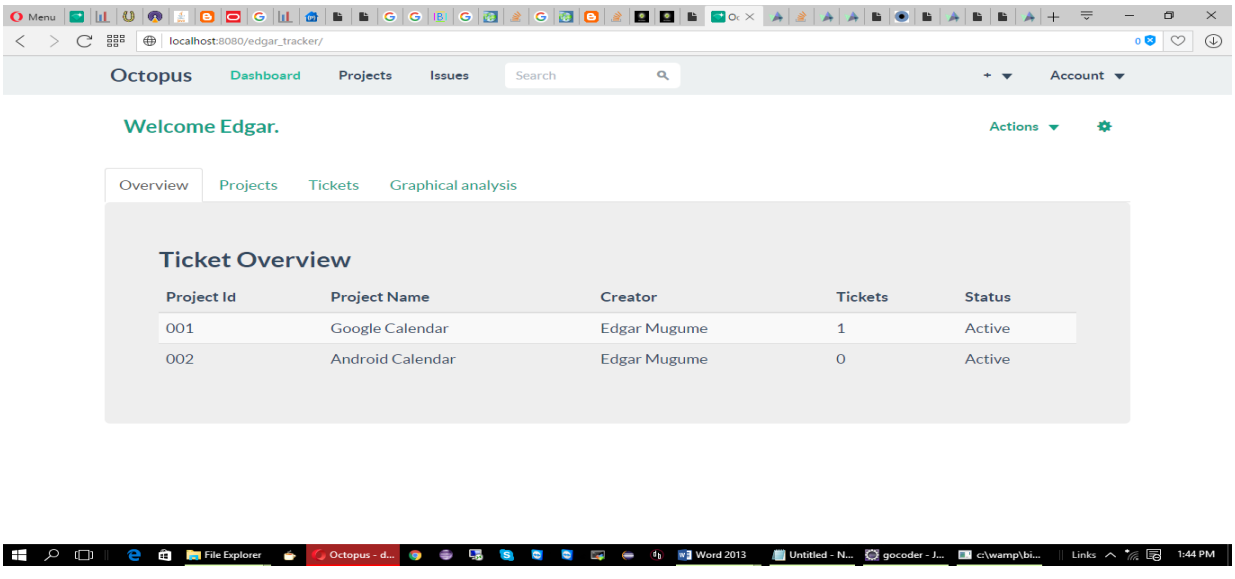

Figure 5.6: Home Screen.

A user is then able to perform various functions as defined by their assigned role. For instance, viewing project details, viewing or commenting on issues, reporting issues or changing status of an issue logged. To create an issue one must have a project on which the issue will be tagged onto. They can either create a project and then attach an issue to it or create an issue on a project they have access to. This issue will be validated and cross checked to ensure it has not already been created or similar issue created. Below is a sample screen showing how to create a project and attach an issue to it respectively.

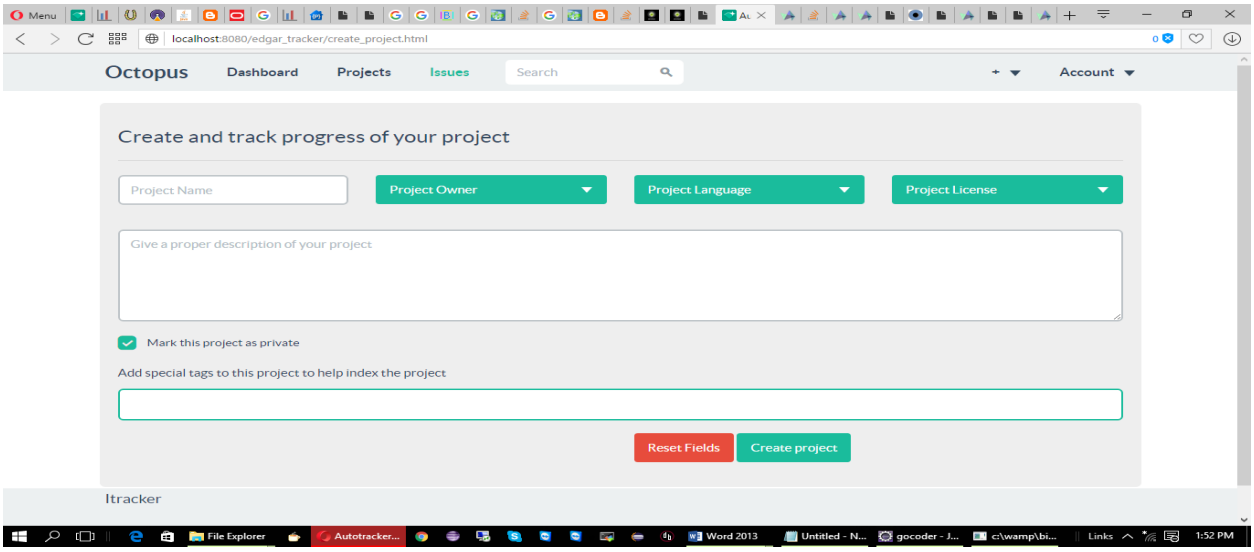

Figure 5.7: Project creation

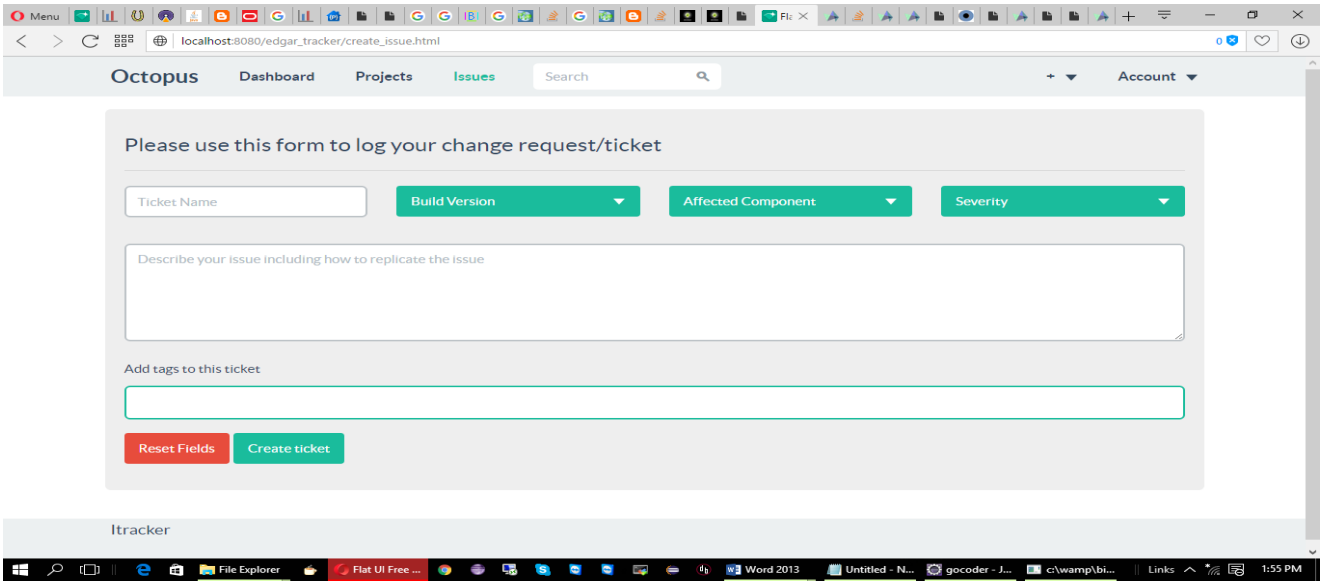

Figure 5.8: Ticket creation

A report can be generated at specific points in the system depending on the user access rights assigned. These reports can be in form of Text file, Excel or PDF as those are the supported formats at the moment. The report display in tabular format also allows for sorting of data per column, searching for specific entries or similar entries in the table. An example is a list of all open projects visible to all users.

5.3.3 Script Description

The Octopus issue tracking system is developed following the object oriented principals and methodologies. It contains 8 directories on the root each containing at least one sub directory and scripts. These scripts support the various JavaScript and CSS functions used in the project. The application logic is precompiled into Java byte code and stored in the WEB-INF directory not accessible to users. Below is a description of the major scripts and directories. Please note that the slash at the beginning of every path denotes the system root and that the application is packaged as a single WAR file or Web Archive file which is unzipped, deployed and managed by Tomcat.

Name	Type	Path	Description
CSS	Directory	/assets/css, /css	These directories contains cascading style sheets. These are files that create the look and feel of the interfaces.
	Directory	/assets/js, /js	These directories contains the JavaScript code to enhance user interface.

	Directory	/assets/img, /img	These directories contain the images used in the system
	Directory	/WEB-INF/libs	This directory is also managed by tomcat and contains all the libraries used by the application and Tomcat.
	Directory	/WEB-INF/classes	Contains the compiled byte code for the Java Virtual Machine to interpret.
	XML Config file	/META-INF/context.xml	This file defines the database connection type and properties. It's used by the application to create a connection pool
	Directory	/wsdl	This directory contains the necessary Webservices definitions for levering the third-party Access APIs.
	Directory	/fonts	This directory contains the custom

			fonts used by the application.
create_issue.jsp	Java Server Page File	create_issue.jsp	Contains the display for creating an issue
create_project.jsp	Java Server Page File	create_project.jsp	Contains the display for creating a project
dashboard.jsp	Java Server Page File	dashboard.jsp	Contains the display for project, issue summary
footer.html	Java Server Page File	footer.html	Contains the display for footer
issue_detail.jsp	Java Server Page File	issue_detail.jsp	Contains the display for issue details
login.jsp	Java Server Page File	login.jsp	Contains the display for login access
project_detail.jsp	Java Server Page File	project_detail.jsp	Contains the display for project details
projects_list.jsp	Java Server Page File	projects_list.jsp	Contains the display for all projects.
projects.jsp	Java Server Page File	projects.jsp	Contains the display for all projects when authenticated
register.jsp	Java Server Page File	register.jsp	Contains the display for registering a new user

search_results.jsp	Java Server Page File	search_results.jsp	Contains the display for search results for anything ranging from project, user, ticket to comment
search.jsp	Java Server Page File	search.jsp	Contains the display for searching for anything ranging from project, user, ticket to comment
ticket_details.jsp	Java Server Page File	ticket_details.jsp	Contains the display for viewing ticket details
tickets.jsp	Java Server Page File	tickets.jsp	Contains the display for all tickets on a project

Table 5.1: Script Descriptions

5.4 Installation of the system

At the site, the system was deployed to the Tomcat Server using the Tomcat Manager App. Tomcat Server was installed as a windows service and started, then accessed via browser using the link http://192.168.2.203:8080 as Tomcat was configured to run on port 80 which is the default HTTP port for internet traffic. The browser used was Opera and the following page was displayed on the tomcat instance.

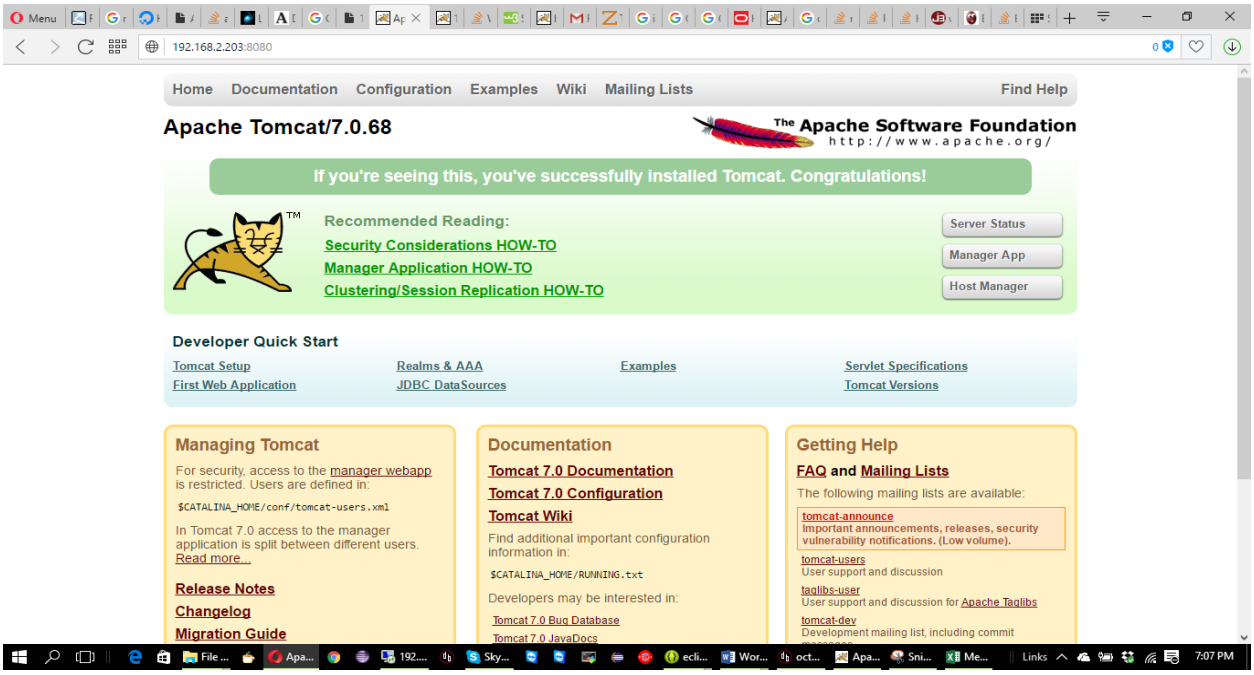

Figure 5.9: tomcat installation page

5.5 System Testing

In order to test the functionality of the system, the prototype was installed on various Operating systems running Java Version 1.7 and above to test system compatibility with the different operating systems. This also helped eliminate major unforeseen bugs and errors and to ensure that the system met all the specified user requirements. The tests carried out include;

5.5.1 Compatibility Testing

Compatibility testing was conducted on the prototype to evaluate the application's compatibility with various computing environments. The system is therefore able to operate on any platform capable of running the Java Virtual machine 1.7 and above thanks to Java's platform independence. The browsers were also tested to see how compatible they can adapt to the Octopus user interface.

5.5.2 Performance Testing

Performance and speed of the application was also tested over a network and scalability also tested. Tomcat has built in load balancers that were able to handle large simulated traffic from over 1000 requests in less than 10 seconds. The Octopus system handled the requests accordingly as dispatched by the Tomcat load balancer.

5.5.3 System Functionality Tests

General Evaluation on System functionality

Access to Octopus (the software issue tracking application) was given to 10 users. 3 developers at Neptune software company limited (system vendor) and 7 IT personnel at centenary bank. These IT personnel included, the IT project Manager, the 6 other IT officers under the core banking system support. They were asked to test and score the performance of the application against a list of predefined attributes. The scale used is as below.

Poor=2	Fair=4	Neutral=6	Good 8	Excellent=10

#	Attribute	Count of Testers Per Attribute Score				
#	Attribute	Poor	Fair	Neutral	Good	Excellent
1	Concurrent Application Access	0	0	0	0	10
2	Graphical User Interface appearance	0	0	0	5	5
3	Application Usability	0	0	0	2	8
4	User Authorization & Authentication	0	0	0	3	7
5	Automatic Issue Assignment	0	0	0	1	8

6	Duplicate bug filtering Capability	0	0	0	2	8
7	Bug Prediction	0	2	3	1	4
8	Change control	0	1	3	3	3

Table 5.2: General Evaluation on System functionality

Summary Grading of Core Functionality

These were carried out, component by component basing on the four core functional requirements and these included; Auto Issue assignment, Duplicate bug filtering, Bug Prediction, Change control

Further, the four core functionalities were tested as a whole system and below is a matrix of marks awarded by the tester. Each function was awarded marks basing on how well it addressed the functional requirement as her been identified by the researcher.

Core Functional Requirement	Auto Issue assignment	Duplicate bug filtering	Bug Prediction	Change control
Percentage Performance Score	80%	85%	65%	75%

Table 5.3: Summary Grading of Core Functionality

Graphical user interface tests were carried out to ensure friendly navigation of the interface by the user across the entire application interface.

CHAPTER SIX

SUMMARY, CONCLUSION AND RECOMMENDATIONS

6.0 Introduction

This chapter presents a conclusion to the system, summarizes achievements and difficulties encountered during the research and recommendations on future work and what others could do as far as this domain is concerned.

6.1 Summary

The core aim and significance of this research was to design and implement a fast, simple and effective issue tracking system with intelligent features such as automated ticket assignment, duplicate filtering, bug prediction and change control management, among others, hence delivering a high performance, lightweight and flexible issue tracking system called Octopus. The need for this arose when it was established and proved that existing issue tracking systems allow reporting of duplicate issues, where a previously encountered and resolved issue could be reported one or more times probably by different users without their knowledge. And that issue tracking systems lacked adequate features to allow automated assignment of issues hence making the issue assignment process manual and time consuming. These systems were also incapable of suggesting possible solutions basing on already resolved bugs. Change control wasn't enriched in these existing issue tracking systems and hence the ability to maintain precise and updated system documentation was insufficient. Furthermore, issue/ bug prediction basing on the recently resolved issues has not been taken into consideration. The researcher then concluded that having the above gaps addressed would significantly improve the timelines in which software issues are solved efficiently and effectively.

6.2 Conclusion

Issue tracking is an important part of every software project and using issue tracking systems is necessary, as it brings on board the following benefits:

a) Improvement in the turnaround time of software issues resolution.

b) Increased user satisfaction and customer appreciation in software issue support.
c) Allows for requests accountability.
d) Greatly improves communication within the team and also to the customers.
e) It also increases the productivity of the team and reduction of operating expenses in software firms.

However, even the best issue tracking system does not help if it does not have a proper request workflow and the responsible team does not learn to report requests correctly. Octopus aims to address these flaws or issues in current issue tracking systems by adhering to the following four basic rules:

1. **Be specific.** A detailed description should state what the awaited result is and what is occurring instead – what one may consider to be a mistake.
2. **Show as much as possible.** If the program writes an error message, it must be copied into the report. Adding screenshots to bug reports as much as possible is very important
3. **Describe how to reproduce the problem. A** programmer will need to reproduce the problem on his or her own computer. After writing down how to do it, the user should go through these instructions him or herself
4. **Clearly differentiate between facts and assumptions.** Sometimes the bug is hidden elsewhere and with assumptions could lead the programmer to a false track. The focus should be on the symptoms and diagnosis should be left to the experts.

Octopus issue tracking system is a young but great software with built in intelligent data processing and manipulation algorithms forming the foundation for its data processing framework makes it a good alternative for companies that need to keep it simple and clean.

6.3 Future Work

Looking back at the research study and the findings that were presented in this paper, we see a study aiming at primarily preventing duplicate bug captures, providing quick fixes and automatic ticket assignment. In future, there is need to have users have more control over the system and also avail a mobile application for the Android, IPhone and Windows mobile platforms.

Only three primary roles of developers, quality assurance personnel, and project managers were sampled for our study. One large area for potential future work then would be to broaden that sampling to include some of the many other stakeholders identified as playing a pivotal role in the development process. Such stakeholders could include customer service and support personnel, customers of the systems being developed that are external to the company, higher-ups within the management hierarchy, or even members of the banking operations staff within the organizations studied.

REFERENCE LIST

ARANDA, J. AND VENOLIA, G. 2009. The secret life of bugs: Going past the errors and omissions in software repositories. In Proceedings of the International Conference on Software Engineering, pages 298–308.

BETTENBURG, R. PREMRAJ, R, ZIMMERMANN, T AND KIM, S,, 2008. Duplicate bug reports considered harmful, In ICSM'08 Proceedings of the 24th IEEE International Conference on Software Maintenance, pages 337–345, 2008

BEYER, H, AND HOLTZBLATT, K 1997. Contextual Design: Defining Customer-Centered Systems (Interactive Technologies), Morgan Kaufmann, San Francisco.

BLACK, R 2002. Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing, Wiley Publishing, New York

BUXTON, B. 2007. Sketching User Experiences: Getting the Design Right and the Right Design (Interactive Technologies). San Francisco: Morgan Kaufmann.

CHURCHILL, E.F, TREVOR, J, BLY, S, NELSON, L AND CUBRANIC, D, 2000. Anchored conversations, In Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems, pages 454–461.

CORBIN, J AND STRAUSS, A 2008. Basics of qualitative research: Techniques and procedures for developing grounded theory, Sage Publications, Los Angeles

DINGSOYR, T AND ROYRVIK, E, 2003. An empirical study of an informal knowledge repository in a medium-sized software consulting company, In Proceedings of the International Conference on Software Engineering, pages 84–92.

HEVNER, A.R., MARCH, S.T., PARK, J. & RAM, S., "Design Science in Information Systems Research, MIS Quarterly, 28(1), 2004, pp. 75-105.

JAANTTI, M, SHRESTHA, A & CATER-STEEL, A 2012. Towards an improved it service desk system and processes: a case study, International Journal on Advances in Systems and Measurements, vol. 5, pp. 203-215.

LYU, M.R, 2011. Handbook of Software Reliability Engineering, McGraw Hill.

ROLAND, M., AND THORING, K 2011. Understanding artifact knowledge in design science: Prototypes and products as knowledge repositories. Proceedings of the 17th Americas Conference on Information Systems.

SALDARINI, R. 2010. Analysis and design of business information systems. Macmillan PC, New York, 2010

TRAJKOV, M AND SMILJKOVIC, A. 2011. A Survey of Bug Tracking Tools: Presentation, Analysis and Trends.

WEISS, C, PREMRAJ, R, & ZIMMERMANN, T 2008. What makes a good bug report? In Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering, pages 308–318.

ZATUL, AS, AND MUDIANA, M 2012, Change and Bug Tracking System, International Journal of Computer Applications (0975 – 8887) Volume 10– No.3

APPENDIX 1

INTERVIEWER QUESTIONING GUIDE

The basic interview protocol was as follows, with additional questions inserted opportunistically based on observations made and participant responses:

1. Can you please give me a tour of the issue tracking system you use?
2. What issue tracking system do you use? Have you always used this system in your current position, or have you switched recently?
3. Can you please describe the primary purpose of issue tracking systems from your perspective?
4. What are the different roles and responsibilities of the people working with the issue tracking system in your department (including yourself)? Do certain people have different levels of access to the issue tracking database than others? a.
    a) Is there a QA process?
    b) Do outside stakeholders (users) have the ability to enter bugs and/or feature requests directly/indirectly into the issue tracking system?
    c) Does management make use of any reports or metrics from the issue tracking system?
5. What is your workflow for managing and resolving bugs/features? What are the steps in the process? Who is/are responsible for each step?
6. What information is usually tracked with each issue?
7. Please walk me through the process of: ("Tell me about the last time you did X...")
    d) Adding a new feature or bug into the issue tracking system.
    e) Starting to work on resolving a bug.
    f) Starting to work on implementing a new feature
8. When working with bugs/features in the issue tracking system, what are the other software tools you use most frequently? Why do you use these particular tools? a. Integration with source code repositories, IDE, email?
9. When reporting an issue, how does the system handle duplicate reporting?
    a) Does it automatically reject and propose existing fixes of similar issues.

b) Does it accept the issue and assign it to the resolver who later learns that it was achieved?
c) Does it allow the issue to be reported and links it with existing similar issues?

10. What is involved in the process of assigning a bug to a user and what is taken into consideration when choosing a resolver to handle the bug?
    d) Experience on similar types of issues?
    e) Amount of work they are currently handling?
    f) Others please specify is available.
11. What types of information do you frequently require from others in order to complete your day-to-day issue tracking and related software development tasks? How do you usually obtain this information? How frequently do you find yourself looking for these kinds of information?
12. What communication channels do you use to communicate with your colleagues and other project stakeholders? How frequently do you use each communication channel?
    a) Instant messaging such as Skype
    b) Micro-blogging (such as twitter)
    c) Chat (e.g., IRC, Skype)
    d) Email
    e) Wiki
    f) Mailing lists/bulletin boards
    g) Verbal conversation (face-to-face, phone/Skype)
    h) Physical notebook
13. Do you have a preference as to which communication channel you use? If so, under what conditions and why? a.
    a) Try to discern motivation for use of one channel over another
    b) Synchronous vs. asynchronous component
    c) Normally archived vs. not normally archived (i.e., availability of communication history to the rest of the team, management, etc.)
14. What shortcomings have you experienced with the tools you currently use from a communication/collaboration perspective? What workarounds have you come up with? Are there any shortcomings that stop you from completing your issue tracking or related

software development tasks? 14. Is there anything else you'd like to tell me about that I haven't asked you about?

APPENDIX 2

Samples snippets of Code in line with the sample screen shots as displayed in chapter five of the main body.

i. Sample Code for Login screen

```
<div class="container">
    <h4>
        <em style="padding-left: 150px">Hey, this is Itracker with a
            capital I</em>
    </h4>
    <div style="padding-left: 150px; padding-right: 150px">
        <div class="login-form"
            style="padding-left: 250px; padding-right: 250px;">
            <h4>Get started today</h4>
            <form action="login" method="post" class="form">
                <div class="form-group">
                    <input type="text" class="form-control login-field" name="user"
                        required="required" placeholder="Username" id="login-name" /> <label
                        class="login-field-icon fui-user" for="login-name"></label> <input
                        type="hidden" name="idkey" value="login" />
                </div>
                <div class="form-group">
                    <input type="password" class="form-control login-field" name="pwd"
                        required="required" placeholder="Password" id="login-pass" /> <label
                        class="login-field-icon fui-lock" for="login-pass"></label>
                </div>
                <div class="form-group">
                    <input type="submit" class="btn btn-primary btn-lg btn-block"
                        value="Login">

                </div>
            </form>
        </div>
    </div>
</div>
<!-- /container -->


<jsp:include page="footer.html"></jsp:include>

```

ii. Sample code for Home Screen.

```java
package org.xmlite.servlet;

import java.text.ParseException;

public class IOMap {

    private static void authenticate(HttpServletRequest request) {
        Sysuser user = new Sysuser();
        user.setUser_name(request.getParameter("user"));
        user.setPass_word(request.getParameter("pwd"));
        user = user.login();
        if (user != null) {
            HttpSession session = request.getSession(false);
            synchronized (session) {
                session.setAttribute("user", user);
            }
            request.setAttribute("tickets", Ticket.loadTickets(user.getId()));
            request.setAttribute("projects", Project.loadProjects(user.getId()));
            request.setAttribute("next_pg", "/dashboard.jsp");
        } else {
            request.setAttribute("next_pg", "/login.jsp");
        }
    }

    private static void processLogoutRequest(HttpServletRequest request) {
        HttpSession session = request.getSession(false);
        if (session != null)
            session.invalidate();
        request.setAttribute("next_pg", "/login.jsp");
    }
```

iii. Sample Code for Project creation

```java
package org.xmlite.beans;

import java.sql.Connection;

public class Project extends BaseDao {

    public Boolean save() {
        try (Connection con = getConnection()) {
            int rows = new Command(con).insert(this);
            return rows > 0;
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return false;
    }

    public Integer update() {
        try (Connection con = getConnection()) {
            int rows = new Command(con).update(this);
            return rows;
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return 0;
    }

    public Integer delete() {
        try (Connection con = getConnection()) {
            int rows = new Command(con).delete(this);
            return rows;
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return 0;
    }
```

iv. Sample Code for Ticket creation

```java
package org.xmlite.beans;

import java.sql.Connection;

public class Ticket extends BaseDao {

    public Object loadAllObjects() {
        Query query = null;
        try {
            query = new Query(getConnection());
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return query.readList(Ticket.class, "select * from opetrack.ticket");
    }

    public static List<Ticket> loadTickets(int id) {
        List<Ticket> tickets = new ArrayList<>();
        try (Connection con = getConnection()) {
            Query query = new Query(con);
            tickets = query.readList(Ticket.class, "select * from opetrack.ticket where creator_id = ?", id);
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return tickets;
    }

    public List<Ticket> loadObjectById(int id) {
        Query query = null;
        try {
            query = new Query(getConnection());
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return query.readList(Ticket.class, "select * from opetrack.project where owner_id = ?", id);
    }

```