Quality assurance is a planned and systematic pattern of all actions necessary to provide confidence that an item or product conforms to established technical requirements. In a competitive market, quality assurance is essential to reduce unwanted cost of rework. Reducing cost by detecting and preventing defects at earlier stages of Software development phases, Software Companies can maximize benefits in different stages of software development life-cycle. This paper focuses on detection and prevention of defects at earlier stages of software development and designing optimum quality assurance practices to make tradeoff between the quality and the cost. Resource wastage and rework in software production can be visible and analyzed thus organization can reach the objective of the best balance between software quality vs cost and maximize net benefit.

Keywords: Software quality assurance, Defect Prevention, Process Improvement, Gross Benefit, Net Benefit

1. INTRODUCTION

To get the real scenario about the software quality assurance [21] practices we visited some software outsourcing company in Bangladesh. These companies are offshore software development and information and communication technology (ICT) consulting firm which develops software product, provides application and web development/solutions and performs IT consultancy in various fields for many businesses in Europe and other parts of the world. These companies define itself by emphasizing central focus on providing best services to valued customers. They offer efficient solutions to valued customers by integrating solutions into their businesses' strategy, practices and tools. Their main focus is to help customers add value to their businesses through the services provided by them. They believe in mutually beneficial long term partnership with their customers and they significantly invest their resources on learning & implementing new technologies in the most innovative manner to enhance performance, promote efficiency and finally, add tangible values to the businesses of our customers.

The focal point of all services provided by these software companies is customer satisfaction and the foundation is quality assurance [21] policy. They believes and practices in creating long term mutually beneficial relationship with customers by establishing close partnership at both technical as well as management level and by understanding the customers’ business focus, values, practices, and processes. Their quality assurance policy ensures that all deliverables provided on time, kept within scopes, delivered with quality as agreed upon by both customers and the outsourcing companies; and thus ensuring value addition to the business of our customers. Since they have the vision “Value Added Off-shore Services” is to add measurable business value for their customers in addition to integrating technology to Off-shore Software Development, they should emphasis on improving research methodology to ensure software quality.

1.1 PURPOSE

The purpose of this document is adhering to defect detection and defect prevention techniques to enhance quality of the product. Pro-active Defect Prevention (DP) is to create an environment for controlling defects and reduce cost. Defects with the ratio of only 80% can be captured by inspection and testing. Cost required for rework found being more expensive than the cost incurred in adhering to DP strategies. The focal point of quality cost investment is to invest in right DP activities rather than investing in rework which had seen as an outcome of un-captured defects.

1.2 SCOPE

This document describes an analysis based on data obtained from leading software companies of varying software production competence. Defect prevention (DP) is a process of identifying defects, their root causes and corrective and preventive measures taken to prevent them from recurring in future. Identified defects classified at two different points in time 1) time when the defect first detected and 2) time when defect fixed. If a defect dwells for a longer time in the product, it is more expensive to fix it. Therefore, it is necessary to reduce defect injection and boost defect removal efficiency. The cost of rework for 1% of defect when identified at the customer’s site is 10 times the cost required for fixing the same defect when identified in-house. As a matter-of-fact, companies adapting to DP strategies over a period of time, quality of the product enhanced while the cost of quality reduced. This document covers all of the activities and support required to reduce cost and reduce rework from the software requirements analysis phase through completion of the system test phase of the software life-cycle. Identifies the defects of the project and the activities, processes, and work products developer will review and audit Identifies the work products.
2.0 METHODOLOGY OF DATA GATHERING AND ANALYSIS

2.1 CHARACTERISTICS OF SOFTWARE QUALITY

Software has both external and internal quality characteristics. External characteristics are characteristics that a user of the software product is aware of including:

- **Correctness** - The degree to which a system is free from faults in its specification, design, and implementation.
- **Usability** - The ease with which users can learn and use a system.
- **Efficiency** - Minimal use of system resources, including memory and execution time.
- **Reliability** - The ability of a system to perform its required functions under stated conditions whenever required—having a long mean time between failures.
- **Integrity** - The degree to which a system prevents unauthorized or improper access to its programs and its data. The idea of integrity includes restricting unauthorized user accesses as well as ensuring that data accessed properly—that is, that tables with parallel data modified in parallel that date fields contain only valid dates, and so on.
- **Adaptability** - The extent to which a system used, without modification, in applications or environments other than those for which it specifically designed.
- **Accuracy** - The degree to which a system, as built, is free from error, especially with respect to quantitative outputs. Accuracy differs from correctness; it is a determination of how well a system does the job.
- **Robustness** - The degree to which a system continues to function in the presence of invalid inputs or stressful environmental conditions. Some of these characteristics overlap, but all have different shades of meaning that are applicable more in some cases, less in others.

External characteristics of quality are the only kind of software characteristics that users care about. Users care about whether the software is easy to use, not about whether it’s easy for us to modify. They care about whether the software works correctly, not about whether the code is readable or well structured.

Programmers care about the internal characteristics of the software as well as the external ones, and it focuses on the internal quality characteristics. They include:

- **Maintainability** - The ease with which we can modify a software system to change or add capabilities, improves performance, or correct defects.
- **Flexibility** - The extent to which we can modify a system for uses or environments other than those for which specifically designed.
- **Reusability** - The extent to which and the ease with which we can use parts of a system in other systems.
- **Readability** - The ease with which we can read and understand the source code of a system, especially at the detailed-statement level.
- **Testability** - The degree to which we can unit-test and system-test a system; the degree to which we can verify that the system meets its requirements.
- **Understandability** - The ease with which we can comprehend a system at both the system-organizational and detailed-statement levels.

The difference between internal and external characteristics isn’t completely clear-cut because at some level internal characteristics affect external ones. Software that isn’t internally understandable or maintainable impairs our ability to correct defects, which in turn affects the external characteristics of correctness and reliability. Software that isn’t flexible cannot enhance in response to user requests, which in turn affects the external characteristic of usability.

The point is that some quality characteristics emphasized to make life easier for the user and some emphasized to make life easier for the programmer.

The following chart shows only typical relationship among the quality characteristics. On any given project, two characteristics might have a relationship that’s different from their typical relationship.
2.2 FINDING A DEFECT

Debugging consists of finding the defect and fixing it. Finding the defect (and understanding it) is usually 90 percent of the work. Debugging by thinking about the problem is much more effective and interesting than debugging with an eye of newt.

2.3 THE SCIENTIFIC METHOD OF DEBUGGING

Here are the steps we go through when we use the scientific method:

i. Gather data through repeatable experiments.
ii. Form a hypothesis that accounts for the relevant data.
iii. Design an experiment to prove or disprove the hypothesis.
iv. Prove or disprove the hypothesis.
v. Repeat as needed.

This process has many parallels in debugging. Here’s an effective approach for finding a defect:

i. Stabilize the error.
ii. Locate the source of the error (the “fault”).
   a. Gather the data that produces the defect.
   b. Analyze the data that has gathered and form a hypothesis about the defect.
   c. Determine how to prove or disprove the hypothesis, either by testing the program or by examining code.
   d. Prove or disprove the hypothesis using the procedure identified in ii(c).
iii. Fix the defect.
iv. Test the fix.
v. Look for similar errors.

2.4 BENEFITS OF EARLY DETECTION AND PREVENTION

<table>
<thead>
<tr>
<th>Table 2.4: Cost of Defects/ Price of quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase</td>
</tr>
<tr>
<td>--------------------------------------</td>
</tr>
<tr>
<td>Definition</td>
</tr>
<tr>
<td>High-Level Design</td>
</tr>
<tr>
<td>Low-Level Design</td>
</tr>
<tr>
<td>Code</td>
</tr>
<tr>
<td>Unit Test</td>
</tr>
<tr>
<td>Integration Test</td>
</tr>
<tr>
<td>System Test</td>
</tr>
<tr>
<td>Post-Delivery</td>
</tr>
</tbody>
</table>

3.0 ANALYSIS OF ACTION, DESCRIPTION AND RESPONSIBILITY

As special technical skills needed, such as those of database administrators, quality assurance [21] specialists, human factors specialists, and technical writers, it becomes more and more important to plan organization structures carefully. Indeed, among the hallmarks of the larger leading-edge corporations are measurement specialists and measurement organizations. One of the useful by-products of measurement is the ability to judge the relative effectiveness of organization structures such as hierarchical vs. matrix management for software projects and centralization vs. decentralization for the software function overall. Here too, measurement can lead to progress and the lack of measurement can lead to expensive mistakes.

The scientific method isn't really one set of methods, but a larger set of guiding principles. It's about developer want to find out how the system works; software testers want to know how the software they're testing works. Those two missions share a lot in common. The scientific method based on observation and experimentation. Testing is the same thing. We set up tests that are very much like experiments, and then we run them and observe what happens. That's the same way scientists test their hypotheses. We run experiments, measure the results and analyze the data to figure out what's really happening. The concept of empirical falsifiability is just proving ideas wrong through experiments. Testing is very similar in that we can't prove the software is flawless; we can only find ways to make the app fail through testing.

If you ask a business manager how much to test the software, they'll probably tell you to test everything. Good testers let them know we can't test everything. It would take an infinite number of tests to get at every possible scenario. We can only look for conditions under which software fails. If tests find no failures, we can have more confidence that it's going to work, but we're still not ever completely sure. After many failed attempts to disprove a hypothesis, scientists build up confidence in hypothesis. It gives their theories credibility. Software testers are really doing the same thing. Because tests are like experiments and they contain many variables in them, software testers should be using what scientists in many industries have been doing for decades namely use smart test design methods that allow them to learn as much actionable information in as possible in each test they run. There is a scientific approach to doing just that. It is called "Design of Experiments." As a result, the tests they construct are highly repetitive of one another and they miss many important gaps in coverage.
Table 3.1: Overview of Software Estimation Steps

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
<th>Responsibility</th>
<th>Output Summary</th>
</tr>
</thead>
</table>
| **Step 1:** Gather and Analyze Software Functional & Programmatic Requirements | Analyze and refine software requirements, software architecture, and programmatic constraints. | Software manager, system engineers, and cognizant engineers. | • Identified constraints  
• Methods used to refine requirements  
• Resulting requirements  
• Resulting architecture hierarchy |
| **Step 2:** Define the Work Elements and Procurements project. | Define software work elements and procurements for specific | Software manager, system engineers, and cognizant engineers. | • Project-Specific product based software WBS  
• Procurements  
• Risk List |
| **Step 3:** Estimate Software Size | Estimate size of software in logical Source Lines of Code (SLOC). | Software manager, cognizant engineers. | • Methods used for size estimation  
• Lower level and total software size estimates in logical SLOC |
| **Step 4:** Estimate Software Effort | Convert software size estimate in SLOC to software development effort. Use software development effort to derive effort for all work elements. | Software manager, system engineers, and software estimators. | • Methods used to estimate effort for all work elements  
• Lower level and Total Software Development Effort in work-months (WM)  
• Total Software Effort for all work elements of the project WBS in work-months  
• Major assumptions used in effort estimates |
| **Step 5:** Schedule the effort | Determine length of time needed to complete the software effort. Establish time periods of work elements of the software project WBS and milestones. | Software manager, cognizant engineers, and software estimators. | • Schedule for all work elements of project’s software WBS  
• Milestones and review dates  
• Revised estimates and assumptions made |
| **Step 6:** Calculate the Cost | Estimate the total cost of the software project. | Software manager, cognizant engineers, and software estimators. | • Methods used to estimate the cost  
• Cost of procurements  
• Itemization of cost elements in dollars across all work elements  
• Total cost estimate in dollars |
| **Step 7:** Determine the Impact of Risks | Identify software project risks, estimate their impact, and revise estimates. | Software manager, cognizant engineers, and software estimators | • Detailed Risk List  
• Methods used in risk estimation  
• Revised size, effort, and cost estimates |
| **Step 8:** Validate and Reconcile the Estimate Via Models and Analogy | Develop alternate effort, schedule, and cost estimates to validate original estimates and to improve accuracy. | Software manager, cognizant engineers, and software estimators | • Methods used to validate estimates  
• Validated and revised size, effort, schedule, and cost estimates |
| **Step 9:** Reconcile Estimates, Budget, and Schedule | Review above size, effort, schedule, and cost estimates and compare with project budget and | Software manager, software engineers, software estimators, and sponsors. | • Revised size, effort, schedule, risk and cost estimates  
• Methods used to revise estimates |
Step 10: Review and Approve the Estimates

- Review and approve software size effort, schedule, and cost Estimates
- The above personnel, software engineer with experience on similar project, line and project management.
- Revised functionality
- Updated WBS
- Revised risk assessment
- Problems found with reconciled estimates
- Reviewed, revised, and approved size, effort, schedule, and cost estimates
- Work agreement(s), if necessary

Step 11: Track, Report, and Maintain the Estimates

- Compare estimates with actual data. Track estimate accuracy. Report and maintain size, effort, schedule, and cost estimates at each major milestone.
- Software manager, software engineers and software estimators
- Evaluation of comparisons of actual and estimated data
- Updated software size, effort, schedule, risk and cost estimates
- Archived software data

4.1 OBSERVATIONS ON THE OUTPUT OF ANALYSIS

<table>
<thead>
<tr>
<th>Objective</th>
<th>Policy</th>
<th>Procedures Practice &amp; Sequence</th>
<th>Standard DOC</th>
<th>Cost</th>
<th>Help Objective</th>
</tr>
</thead>
</table>
- Assure internal reviews to eliminate ambiguities and uncertainties
- Assure internal reviews to define requirements in terms of their testability |
### Table 4.2: Current Capability Assessment about DESIGN

<table>
<thead>
<tr>
<th>Objective</th>
<th>Policy</th>
<th>Procedures Practices &amp; Sequence</th>
<th>Standard DOC</th>
<th>Cost</th>
<th>Help Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>THE DESIGN considerations must cover any assumptions or dependencies which need to be addressed or resolved before attempting to devise a complete design solution.</td>
<td>Architectural Design</td>
<td>Project Schedule Design Document Review Checklist Meeting Minutes Data Dictionary Entity Relationship Diagram User Interface</td>
<td>Design Document Review Check List Configuration Plan Configuration Control Architectural/Logical phase of Designing</td>
<td>$2009</td>
<td>- Promote peer inspections of new/modified design components for new releases - Assure proposed design changes are approved</td>
</tr>
</tbody>
</table>

### Table 4.3: Current Capability Assessment about CODING

<table>
<thead>
<tr>
<th>Objective</th>
<th>Policy</th>
<th>Procedures Practices &amp; Sequence</th>
<th>Standard DOC</th>
<th>Cost</th>
<th>Help Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coding procedure will maintain for both new project and enhancement of existence software project/module</td>
<td>Company Code Convention Source Code Tagging Configuration Management Plan Review Meeting</td>
<td>Review the details design plan, identify legacy/external resources consider development and language tools technology selected forcing before start the actual coding phase by a review meeting, maintain the Project Schedule accordingly</td>
<td>The default java &amp; Microsoft coding convention is used computing and software infrastructure</td>
<td>$10000</td>
<td>- Review code against coding standards (source lines of code, complexity)</td>
</tr>
</tbody>
</table>

code review meeting Minutes of the review is prepared and kept in the Project File Approval for commencing Testing Phase is given.

### Table 4.4: Current Capability Assessment about TESTING

<table>
<thead>
<tr>
<th>Objective</th>
<th>Policy</th>
<th>Procedures Practices &amp; Sequence</th>
<th>Standard DOC</th>
<th>Cost</th>
<th>Help Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>To ensure that the development is complete as per the Requirement of the Client for both new software project and for the maintained software project.</td>
<td>Purpose and Scope Methodology Risk Analysis Configuration Plan Test Plan Progress Monitoring Project Schedule Deliverables Team Structure</td>
<td>Test Planning is done during the Analysis Phase and is stated in the Project plan Unit testing System acceptance security test &amp; other tests Test Plan Test Schedule Test Specifications</td>
<td>Functional Tests Boundary Tests Performance Tests</td>
<td>$4000</td>
<td>- Participate in dry runs to assure real-time performance - Monitor actual timing results during stand-alone, integration and system level testing</td>
</tr>
</tbody>
</table>
Table 4.5: Current Defect Detection Assessments

<table>
<thead>
<tr>
<th>Defect</th>
<th>Detection</th>
<th>Case Analysis</th>
<th>Design</th>
<th>Coding</th>
<th>Testing</th>
<th>Cost of Defect Fixation</th>
<th>Migrant Cost</th>
<th>Cost of Prevention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ambiguity</td>
<td>Redundancy</td>
<td>Code Logic</td>
<td>Feedback from customers, staff, project reviews</td>
<td>Performance and testing</td>
<td>Properly maintaining documentation and process improvements</td>
<td>Automated testing standards for testing, user interface</td>
<td>$2500</td>
<td>$2500</td>
</tr>
</tbody>
</table>

5.0 Suggestions for improving better balance between quality and cost based on analysis

In the previous tables we have certainly observed that prevention of defects and detection of early defects is the major requirement to improve software quality. If the error detected at later stages the cost is also increasing proportionally in order to fixing the bugs. Even the quality decreases if the errors are detected at later stages because fixing a bug at later stages may add another bug and cause system malfunctioning.

Based on the scenario we shall propose for improving better balance between quality and cost based on analysis are as follows:

Table 5.1: Proposed Capability Assessment about REQUIREMENTS

<table>
<thead>
<tr>
<th>Objective</th>
<th>Policy</th>
<th>Procedures Practices &amp; Sequence</th>
<th>Standard DOC</th>
<th>Cost</th>
<th>Help Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gathering eliciting ascertaining uncovering requirements</td>
<td>Formulate specification techniques defensive design</td>
<td>Focus on interfaces between the software and the system in analyzing the problem domain identify critical hazards early in the requirement analysis</td>
<td>Function Identification Function Organization Function Specification Functional Requirements Documentation Requirements Performance Requirements Security Requirements Interface Requirements Portability Requirements Resource Requirements Maintainability Requirements Acceptance Testing Requirements</td>
<td>$1500</td>
<td>-Assure use of the requirements volatility metrics - Maintain system requirements - Assure functional baseline</td>
</tr>
</tbody>
</table>
Table 5.2: Proposed Capability Assessment about DESIGN

<table>
<thead>
<tr>
<th>Objective</th>
<th>Policy</th>
<th>Procedures Practices &amp; Sequence</th>
<th>Standard DOC</th>
<th>Cost</th>
<th>Help Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design consists of multiple views (both static and dynamic)</td>
<td>Hierarchical decomposition</td>
<td>Project Schedule Design Document Review Checklist Meeting Minutes Data Dictionary Entity Relationship Diagram User Interface</td>
<td>Data Flow Diagrams Transformation Schema Structured English Decision Tables State-Transition Diagrams Transition Tables Precondition-Post conditions</td>
<td>$1500</td>
<td>Participate in formal customer design reviews with the customer. ASSure allocated baseline. Assure that test procedures cover all testable requirements.</td>
</tr>
<tr>
<td>A design is evaluated against goals (requirements) using standard properties (e.g., coupling and cohesion)</td>
<td>Top-down design Object-oriented design Functional design</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3: Proposed Capability Assessment about CODING

<table>
<thead>
<tr>
<th>Objective</th>
<th>Policy</th>
<th>Procedures Practices &amp; Sequence</th>
<th>Standard DOC</th>
<th>Cost</th>
<th>Help Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formatting, Layout and Style Defence Programming Managing Construction</td>
<td>Commenting Format of Control Structures Compound Statements switch case Statement</td>
<td>naming convention must be followed performing activities and their sequence to comply policies</td>
<td>The default java &amp; Microsoft coding convention is used comparing and software infrastructure Source code for functions should</td>
<td>$8500</td>
<td>Assure internal SCM for problem control and corrective action logs. Assure version control of development software prior to integration.</td>
</tr>
<tr>
<td></td>
<td>Entry Condition Loops vs. Exit Condition Loop Functions</td>
<td>generally not exceed 50 lines of code. Functions shall begin on a new page.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.4 Proposed Capability Assessments about TESTING

<table>
<thead>
<tr>
<th>Objective</th>
<th>Policy</th>
<th>Procedures Practices &amp; Sequence</th>
<th>Standard DOC</th>
<th>Cost</th>
<th>Help Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>The goal testing for software is to quickly find the defects in requirements and code and get the software running as an integrated component of the enterprise financial system as well as provide guidance for the personnel testing the software.</td>
<td>Purpose and Scope Methodology Risk Analysis Configuration Plan Test Plan Progress Monitoring Project Schedule Deliverables Team Structure Testing of individual program components Defect testing</td>
<td>Test Planning is done during the Analysis Phase and is stated in the Project plan. Unit testing System acceptance security test &amp; other tests Test Plan Test Schedule Test Specifications</td>
<td>Acceptance testing Alpha test Beta test Installation testing Testing process goals Validation testing System testing Component testing Comparison testing Compatibility testing End-to-end testing Risk analysis testing Regression testing Compliance testing</td>
<td>$3000</td>
<td>Assure adequate regression testing as necessary. Assure adequate description exists of the released software version.</td>
</tr>
</tbody>
</table>
### Table 5.5: Quality and Cost Benefit Based Analysis of proposed capability:

<table>
<thead>
<tr>
<th>Entry</th>
<th>Defect</th>
<th>Case Analysis</th>
<th>Requirement</th>
<th>Design</th>
<th>Coding</th>
<th>Testing</th>
<th>Cost of Defect Fixation</th>
<th>Migration Cost</th>
<th>Cost of Prevention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Standard</td>
<td>Redundant code</td>
<td>Training on Database structure and object model Perfom causal analysis and prioritize root causes Identify and develop solutions</td>
<td>Pact analysis Implement solutions Review the status and benefits of DP at end of iteration</td>
<td>Do a requirement walkthrough Improve the communication and coordination Should be examinr concurrency problems or time</td>
<td>Define the rules and procedures for realizing an architecture in a completed system</td>
<td>Review the status and benefits of DP at end of iteration</td>
<td>Functional Database design Evaluate the architecture implemented</td>
<td>Logical flow maximize size and complexity of procedures error handling</td>
<td>Validation testing Defect testing Unit testing</td>
</tr>
</tbody>
</table>
6.0 RECOMMENDATION TO IMPROVE

6.1 IMPROVE PROJECT SQA PROCESSES

The SQA activity for process improvement requires:

I) Understanding project and SQA processes
II) Determining where inefficiencies or defects occur (root causes of defects)
III) Recommending changes to project processes to improve efficiency or reduce defects
IV) Recommending improvements to eliminate the root causes of defects
V) Recommending training courses for the project team

The purpose of this activity is for SQA to review existing project and SQA processes and report on efficiencies and areas for improvement and identify processes that need to define. To improve project SQA processes, SQA needs to review and audit both project processes and SQA processes. This will ensure that project processes and project SQA processes consistent and compatible with one another. Process improvement may result in changes to the policy, processes, and/or procedures.

6.2 Measurements for Defect Analysis

In some sense the goal of all methodologies and guidelines is to prevent defects. For example, a design methodology gives a set of guidelines that if used will give a good design. In other words, the design methodology aims to prevent the designer from introducing design defects by guiding him along a path that produces good and correct designs.

However, by defect prevention (DP) we mean learning from actual defect data from a project with the goal of developing specific plans to prevent defects from occurring in the future. As the main goal of DP is reduction in defect injection and consequent reduction in rework effort, it is best if suitable measurements made such that impact of DP can quantitatively evaluated. That is, a project employing DP should be able to see the impact of DP in the injection rate and on the rework effort on the project. For both of these proper metrics have to be collected. Furthermore, suitable data needs to be collected to facilitate the root cause analysis for DP. The measurements needed for evaluating the effectiveness are defects and effort. For defects, data on all the defects found and their types needed.

This data is easily available if projects follow the practice of defect logging, as is the case in most mature organizations. To facilitate defect analysis, for each defect, its categorization in a fixed set of categories should also record. A classification like the one proposed by the IEEE standards [23], or by the orthogonal-defect classification scheme [22] can be used.

Frequently, organizations log information like detection stage, injection stage, etc to facilitate different types of analyses. Details about the different parameters recorded during defect logging given in [9]. For understanding the impact of DP on rework, the effort spent on the project needs to record with suitable granularity such that rework effort can be determined. Specifically, for each quality control activity, the rework effort should not club together with the activity effort but must record separately. Effort logging generally requires that each member of the project team record the effort spent on different tasks in the project in some effort monitoring system frequently, different codes used for different categories of tasks and for most of the major tasks the effort divided into three separate categories – activity, review, and rework. With this type of categorization, rework effort for each phase can be determined. Details about the system and codes used for effort reporting mentioned here [9].

These measurements about defects and effort are sufficient to do defect analysis and prevention, as well as quantify the impact of DP. Note that DP can done, and its impact on the defect injection rate can be determined, even if the effort data is not available. However, without the effort data, the impact of DP on rework cannot be determined.

7.0 Cost benefit analysis

Costc of Practicing Current Process
Costim of Practicing improved Process

Cost increase = Costim - Costc
= $1000 - $1500
= $500

Gross Benefit = [CDFc – CDFim + MCc – MCim + CPim - CPc]
= $2500-$500+$2500-$500+$1000-$2000
= $3000

Net Benefit = Gross Benefit - Costim - Costc
= $3000- $500-$1500
= $1000
8. Conclusion and recommendations

Defect prevention can improve both quality and productivity. If the number of defects injected reduces, then the quality improves as the number of residual defects in the delivered software reduces. Furthermore, if we inject fewer defects, fewer defects need to be removed at earlier stage, leading to a reduction in the effort required to remove defects. The subjectivity of Net benefits vs process improvement graph measures the visibility on defection and prevention of defects at earlier stages. Optimum Software quality assurance practices and reduce rework for cost benefit oriented analysis can be visible and analyzed thus organization can reach the objective of the best balance for improving quality product and cost reduction process.

9. References


[23]. IEEE, Std. 1044-1993. IEEE standard definition, classification for software anomalies, IEEE