A Proposed Model for Code Quality Maturity Model

Fariha Motherudin*, Noor Ezleen Md. Moksen
Information Communication and Technology Division, MIMOS Berhad

*Corresponding author. Email: fariha.motherudin@mimos.my.
Manuscript submitted September 12, 2014; accepted January 15, 2015.
doi: 10.17706/jsw.10.3.374-383

Abstract: The information technology (IT) industry is characterized by rapid innovation and intense competition. A key issue is the ability of the development team to leverage emerging software development best practices. It is a critical enabler for the overall health and sustainability of an organization's ability to build software solutions. In this paper, we are proposing an evolutionary model that could be a reference for others in the area of utilizing best practices in the area of code and build management. The best practices studied in this paper are included static code analysis, automated unit testing, continuous integration, release gate checking and technical debt. The proposed model is expected to help appraising the maturity level of the produced codes, build and release management practices. It aims to satisfy the characteristics defined at five levels and estimate the target level that one try to achieve. The proposed model is based on several years of real world experience with a big number of teams from the software engineering field. The classification level is based on practices that practitioners observed during the software development life cycle. The processes have evolved from an independent process using an independent tool to prediction process using technical debt ideas which hope this proposed model would be beneficial not only to the software engineers but also to the organization as a whole.

Key words: Auto deployment, code quality, continuous integration, governance, quality gates.

1. Introduction

Static analysis for source code evaluation, verification and analysis is gaining in popularity due to the wide selection of commercial ready tools that are becoming more developer friendly, low cost and having solutions that are well integrated with the development environment for popular programming languages. The more advance tools provide the organization with a platform for which various tools such as code analyzers, plug-ins and rule sets or libraries for coding standards and convention can be integrated and source code quality information consolidated into a single Code Quality Management dashboard.

Being a mature organization, we recognized the potential of these tools has embarked on a journey to establish a process infrastructure that leverages the capability of such tools for dynamic analysis using unit testing, debuggers and various static code analyzers. The paper outlines a code quality initiative that has evolved from using standalone applications to a solid process infrastructure to realize best practices in quality assurance and management, requirements engineering and source code development principles to achieve higher standards in the software deliverables.

This paper begins with an introduction to the proposed model. Section 2 describes in detailed of each model; the generic practices, characteristics and tools involved. The much detailed information on each level would be available in subsequent sections. Data which has been taken from actual projects are presented in graphical figures for ease of understanding of the differences between the levels. The paper is summarized with a

discussion on the evolution and its impact to all groups in a software development project.

2. Code Quality Overview

The code quality process consists of a set of best practices which include checking for code compliances, code quality, imposing automated unit test that benefits from the continuous integration process.

The main objective is to gauge the level of code quality before release, which concluded in a centralized dashboard. The level of code quality is not merely about the complexity of the code, the tools or the metrics, but it includes the automated quality governance methods in ensuring that the developer's practices are consistent and conformed with the suggested best practices, whichever applicable. By governing the processes, it is expected to produce consistent outputs by the teams and optimistically help to reduce number of technical issues during the quality assurance gates.

It also serves to ease software quality tracking of multiple releases. Besides that, the process will give benefits to design, refactoring by considering the issues and suggestions analyzed by the tools. The important practices that mandated to the project teams in our environment are static analysis, automated unit test and refactoring. This continuous practice would be clearly seen if applied continuous integration process.

2.1 Static Analysis

Static analysis helps to analyze a software program without actually executing the program [1]. Comparing with dynamic analysis, static analysis analyses the exact source code line while dynamic analysis analyses the program while it is being executed; by performing analysis of all kinds of tests on the object codes and executable [2]. In summary, static analysis tools help

- To detect possible bugs which the debugger is unable to detect [1]
- To provide vulnerability and remediation information that is seen as a worthy guidance, especially to the new engineers in upgrading their coding skills
- To adhere coding standard, which maintains ancconsistency across projects

Static analysis and automated unit test are performed using independent tools that integrate with the developer's environment. There are many plugins available either proprietary or community versions, divided by the coding languages used. Some of the examples are available in Table 1 below.

Table 1. Tools Example

Practice	Java	C/C++	C#	PHP
Static Analysis	PMD, Checkstyle,	Gprof, Oprof, Valgrind	Purify	PDepend, CodeSniffer
Unit test and coverage	Junit, Cobertura, EMMA	Cunit, GCov	NUnit, VSTS	PUnit, PMD
Build	ANT, Maven	VS	VSTFS	-

However, these tools when used in a standalone manner are limited in their capabilities. Open platform systems such as Sonar have features that allow integration of various source code analyzers either static or dynamic. Organizations are able to leverage the capabilities to build a quality dashboard with custom code quality metrics that is transparent to both developers and management.

2.2 Unit Test

The second practice that is used in this research paper is a unit test process. A unit test is expected to be conducted after code development completion to test each individual units of source code. This is a classic way of conducting unit test. While in Agile methodology, the approach might be slightly different where the unit test

is conducted before writing the production codes. However, the objective of the process remains the same, which are:

- To determine that it is working as expected
- To improve effectiveness of the screening processes in the software engineering life cycle in order to reduce the number of escaped defects
- To find errors in single component early, instead of finding multiple errors in multiple components which is exponentially more difficult to resolve
- To determine a quantitative measure of unit test success or failure, which is an indirect measure of internal testing effectiveness

Performing a unit test will incur project effort as it requires a learning process before one would excel in writing unit test codes. The team needs to practice writing unit test cases by familiarizing with the concept and rules for specific programming languages. This consumed time and effort in the initial stages after many attempts and project iteration. So to overcome this hassle, some of the suggestions recommended by the industry are targeting at the most complex functions in order to discover coding errors.

2.3 Continuous Integration

Codes are stored in a centralized version source control system which the codes are built continuously by applying continuous integration process. Continuous integration is a practice in merging all software development, working copy with a shared mainline frequently in preventing integration problems, thru automated builds. Each integration is verified by an automated build to detect integration errors as quickly as possible. It starts when a developer commits changes to the source code repository. The build server executes the master build script or delegate execution to another server. The team is notified of building results through a feedback mechanism. If a code fix is needed, developer commits the corrected code back to the repository. This action kicks off a new build cycle.

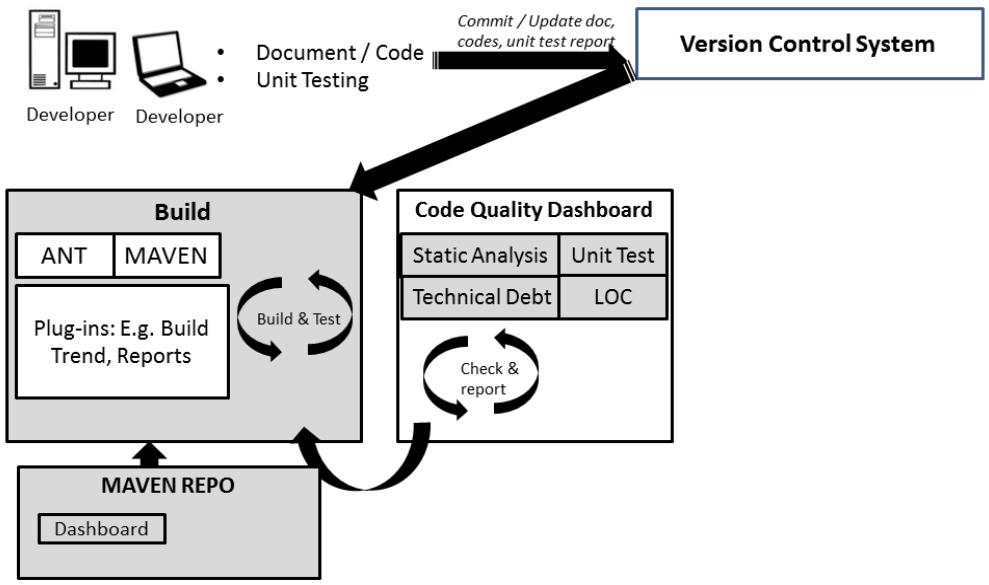

Fig. 1. Architecture view of the tools used in a project.

Fig. 1 above explains on the design view of the mentioned practices. Each tool is interconnected with each other by configuring the appropriate setup and the plugin.

By performing the above practices, developers are expected to refactor their design by referring to the report produced by the tool. The report consists of metrics and recommendations on fixing the errors. While developers improve codes which translated into the metrics results, the quality assurance group would continue

to monitor the code quality progress by observing common trends across projects and identify appropriate action plans [4].

2.4 Quality Gate

The quality assurance monitoring part may be done by the use of quality gates. Quality gates have been used in a variety of industries in order to increase quality of the product and deliverables. Software quality gates, which are normally implemented as entry and exit criteria [4] between phases in the software development lifecycle. In this paper, the quality gate that will be mentioned in the later section is focusing on later stages of software development project which are the test and release stage. Usually the quality gates are done manually thru checklists and require manual checking. This paper is proposing an automated way of implementing the quality gate through automated tools in the context of continuous integration environment.

All the above processes that include code quality and build management practices are summarized into our proposed code quality evolutionary model. It is objectively aimed to secure the four focus areas; code, build, change and release management area for a solid decision making when it comes to a software release. For easier traceability, we divided into five levels. Each level has its general practices to be satisfied that related to the mentioned four process area. The level provides a way to measure how well the practitioners can and do improve their practices. The detail generic practices and expectations for each level are explained in the subsequent sections.

3. Characteristics of Each Level

Levels are used in this paper to describe an evolutionary path recommended for software projects that wants to improve the practices to develop and maintain product quality and its services. Levels characterize improvement from an ill-defined state to a state that uses quantitative information to determine and manage projects in order to meet project datelines.

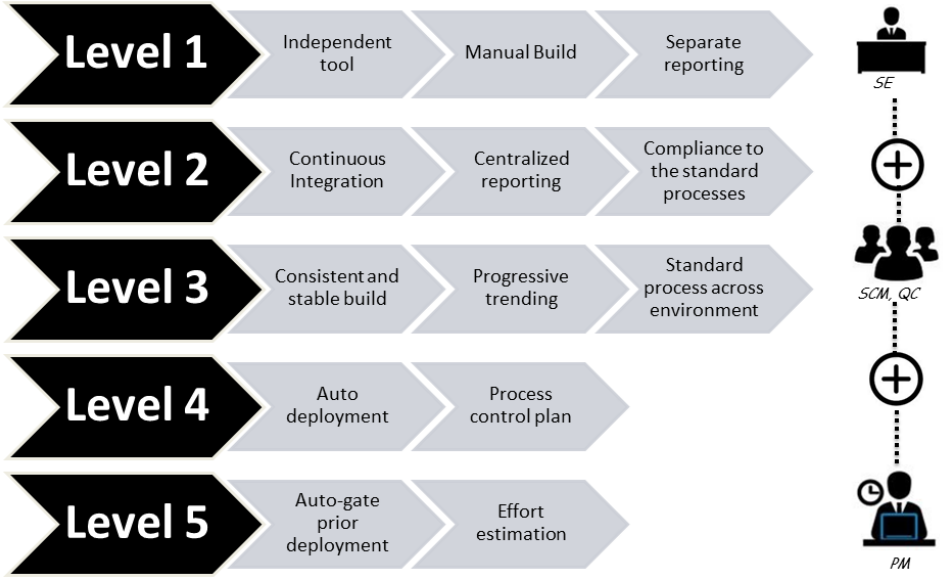

Fig. 2. Code quality maturity proposed level.

Fig. 2 above is an illustration of each level in term of its characteristics and actors involved. At the lower levels, the software engineer (SE) plays an important part in the model. At intermediate levels, the software configuration managers (SCM) and Testers contributes to the success. Once all practices are satisfied, the results are useful for project managers (PM) to perform effort estimation for project scheduling, costing and service delivery.

In summary, the characteristics of each level are briefly explained in Table 2 below. While the detail

explanations are described in each subsequent level's section.

Table 2. Overview of the Levels

Level	Main Characteristics
1	Independent Tool Manual Build Separate Reporting
2	Code Governance Continuous Integration Centralized reporting Standard complied processes
3	Build in a timely manner Metrics collection and monitoring Standard process across environment
4	Auto deployment Trending results
5	Auto gate prior deployment Effort estimation

3.1 Level 1

At level 1, practices are usually performed either fully or partially performed. The team is usually not having a stable environment to support the practices. Success in the projects in fulfilling the practices depends on the competence and heroics of the people in the organization and not on the use of the consistent practices. Level 1 is characterized by a tendency to abandon the practices in a time of crisis and inability to utilize the tools used to repeat their successes.

There are many static analysis tools available for developers to help them to review their codes and select applicable patterns [5]. At this level, the tool is seldomly used or if used, the practice is not applied consistently and not integrated with other tools. Each tool produces separate reporting as it will run individually, and triggered by the developers. It is up to the developers' preference to fix the bugs and resolved the issues as there is no formal mechanism in term of process and enforcement at this point of time. As a result, there is inconsistent use of powerful tools in the practice.

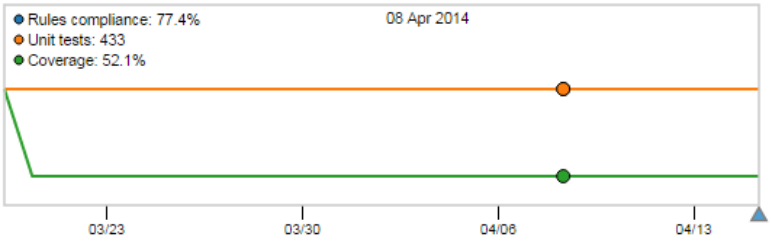

Fig. 3. Sample project 1.

Fig. 3 shows the code quality has been abandoned even though the code analysis tool was used. Four subsequence builds were static with no improvement.

3.2 Level 2

A level 2 is characterized as 'Define' stage where the practices are expected to be consistent and centralized. Enforcement of the code quality and consistency is established. The initial focus of software code governance was to assure software quality and security of in-house developed code by establishing clear guidelines and procedure [6]. Continuous Integration is fully practiced in assisting build management.

It is expected that connection to various essential systems; source, build, test and deploy are configured and setup at a centralized location which accessible by the project teams. This automated and integrated infrastructure promotes ways in order for the developers to write better quality codes, improve their coding

proficiency levels and produce solid codes and libraries. It establishes an automated way to share information, including audit trail for compliance and measure improvement.

At this level, software configuration manager plays an important role in setting up the infrastructure and monitors the builds. Any build break to be reported back to the developers for fixing. Only success build is promoted to the next iteration.

From the centralized and standard practices at this level, quality control and assurance team would make use of the advantages of this standard process. Reports could be collected to monitor trending and compliances. The sample of the reports is shown in Fig. 4 and Fig. 5 below.

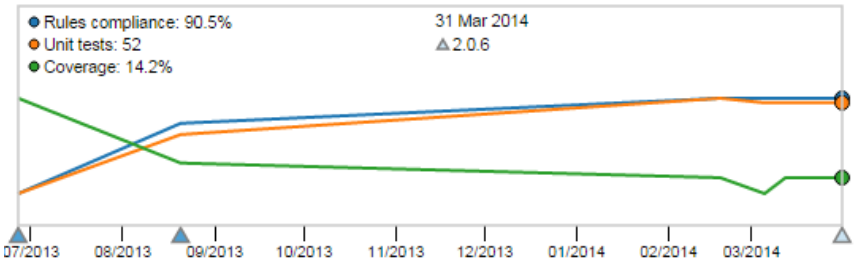

Fig. 4. Sample project 2.

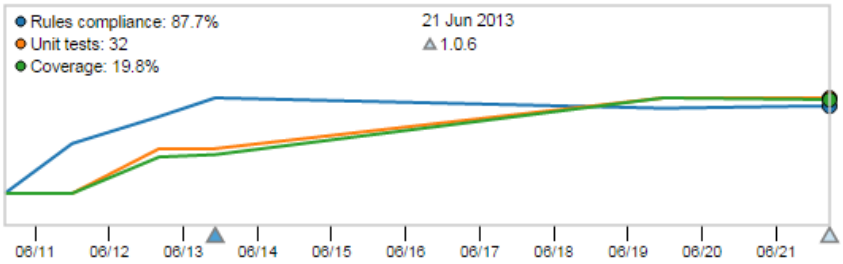

Fig. 5. Sample project 3.

Fig. 4 and Fig. 5 above track the project metrics trending over time. The teams complied to the standard processes which can be seen by the looking at the build events that are consistently appearing. Rule compliance showed improvement over time. There was increasing number of the unit test cases. However, the coverage is still low which means there are still uncovered codes that are not tested. This low coverage also cause by the increasing number of code lines, but not much increment of unit test cases. At a later time of the project, the metrics results are quite constant. This might due to not much development work involved during system test and release phase.

They are meeting Level 2 practices by performing continuous integration process which can be seen from the report that is accessible to all and the consistent integration builds that being recorded in the graph. However, the team will need to improve on the coverage by writing more unit test cases and perform internal screening to improve the product quality in order to achieve project goals.

3.3 Level 3

At level 3, practices are well characterized and understood by the project members. These standard processes are used to establish consistency across the team. A core team of mentors is available to assist in guiding in setting up the tools, build best practices and adopting improvements. They practice daily builds which has been highly recommended by the Agile practitioners. Continuous builds are to ensure that the parts fit together as work is done. It will check whether the changes made have broken anything and making sure they still work. With a centralized reporting, they could monitor the progress that should be a progressive trending especially on the project metrics.

A critical distinction between level 2 and 3 is the trending report should be progressively improved as the practices are managed more proactively using an understanding of the interrelationships of the practices. The team is matured in improving product quality thru the above practices.

The developers and software configuration manager are expected to work together if any incidents or code conflict occurred. At this level, quality control and assurance group monitors the report and identify any action needed in order to improve the processes. In a typically maturity model, the common cause is identified at Level 4. Sample reports are available in Fig. 6 and Fig. 7 below.

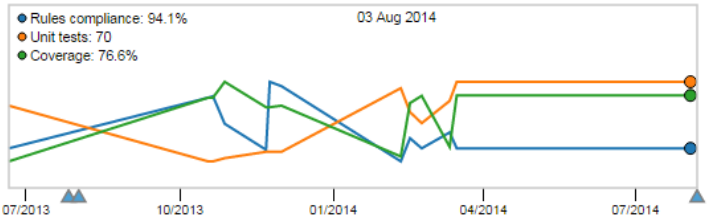

Fig. 6. Sample project 4.

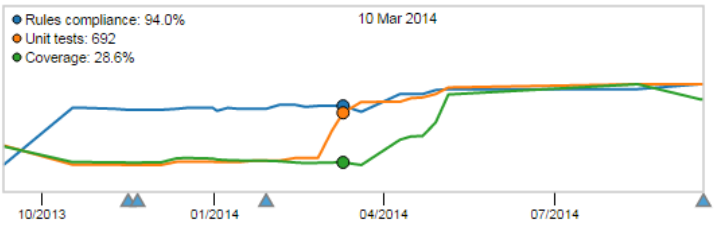

Fig. 7. Sample project 5.

Time series charts above were plotted over two years time. From the charts there are consistent builds over time. The trending is showing positive trends which the team is continually improving the product by writing more and more test cases in order to achieve higher coverage. Although there are hiccups which we believe is due to the higher number of lines being introduced at a certain time of point, they managed to recover and continuously increasing the results. Rule compliance is constant over time due to the team are familiar with the standard rules.

3.4 Level 4

At level 4, the team has already established quantitative objectives for process performance and uses the results as criteria in managing projects. The practices are institutionalized across the projects. The team has matured and any feedbacks are welcome to improve the standards.

Software delivery, integration and build are performed routinely where auto deployment may help to assist the process. During a lower level, all releases are managed manually through face-to-face meetings or informal discussions. At this level, it is expected that all application deployment, middleware configurations and infrastructure are managed centrally in a collaborative environment that leverages automation to maintain the status of individual applications.

Developers, testers and deployment team capable to continuously build, provision, deploy, test and promote automatically.

SCM is required at the point of starting up the auto deployment architecture and only involves when it comes to the related configuration issues.

Auto deployment, seeks to speed time by quickly testing applications and promoting them to subsequent test environments. Recurring deployments can be configured to deploy application versions, for a specific phase of the environment, at a specific time. Scheduled deployments which do not meet the criteria, are flagged as a failure.

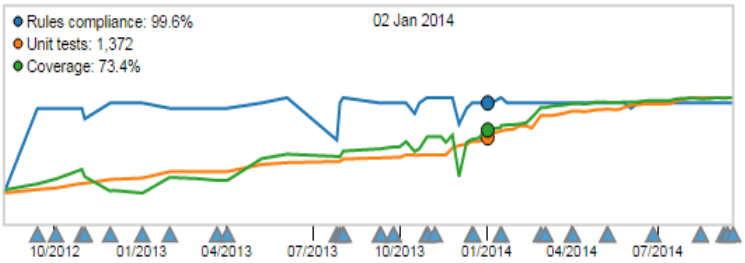

Fig. 8. Sample project 6.

Fig. 8 above showed the project metrics performance that's been collected over the two years. The results are quite stable, with upwards trending. It shows improvement from time to time, and at the later stage of the project, they have achieved a stable performance and achieve the project goals. The threshold is then determined based on the variability between the projects. This will become the right indicator for a given quality aspect and development environment.

3.5 Level 5

At level 5, the focus is on continually improving process performance through incremental, innovative technical practices.

Organization, or the project could specify the conditions that must be met before the application could be promoted in an environment by establishing quality gates [9]. Auto quality gating is performed to gauge visibility into deployment via audit trails. Quality gates are defined at the environment level where it is categorized into a single gate and or conditionals. Quality gates provide a mechanism to ensure that [10]:

- Component versions cannot be deployed into environments unless they have passed the gate-specified status.
- Applications reflect the agreed upon quality criteria for promotion.
- Applications that advance have auditable approvals.

The quality gates also comprise the framework for technical debt assessment. Any omission of these aspects inevitably leads to an incomplete view of the technical aspects concerned and will result in a failure of quality gates.

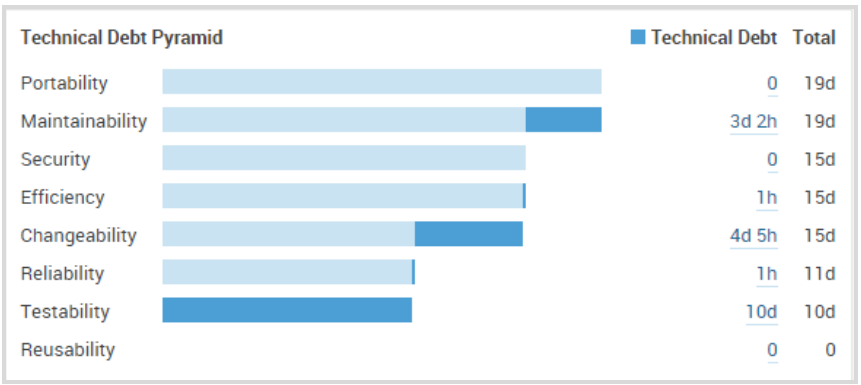

Fig. 9. Sample of technical debt.

Technical debt is a recent metaphor referring to the eventual consequences of poor system design, software architecture. If the debt is not repaid, it will keep on accumulating interest where in the context of software development process is in accumulated by a number of man-days [7]. The higher technical debt means the more time needed to fix the issues. Some arguments being made that the major cause of technical debt are schedule pressure [8]. Software developers tend to make mistakes and ignore the current errors, in order to rush for the datelines.

Technical debt is helpful to be calculated by an automated tools where it defines against a set of errors and number of days to fix the errors. The recommendation is not only seen as useful to software developers, but also

to project managers in estimating project plans. A sample of the report is shown in Fig. 9 below.

Having a predicted number of days required for the rework allows for better management of resource utilization and estimation of development work completion. In certain circumstances, some of the non critical errors might be postponed to a later phase as long the errors are make-known in ensuring the stability of the product.

4. Conclusion

We have proposed a model that helps to gauge the maturity level of the code and build practices. The model is based on studies that has observed an evolution of independent use of code analyzers, unit testings from standalone desktop tool to an integrated architecture using continuous integration process to a centralized gating system for real time monitoring. The model consists of five levels, which aims to satisfy the characteristics of each level. One of the objectives when one use this model is to estimate the target level that one try to achieve. Historical data are presented in each level to differentiate and useful in understanding the maturity level of coding practices. The aim of the practices not only beneficial to software developers, but also aim to help project managers in estimating the effort of future activities.

At the highest maturity level, it is illustrated that every process are automated and require minimal human interventions. Automated process via the tools provide a fast and transparent results to allow organizations to objectively grade and rate project's quality.

We expect that this proposal is beneficial for software development teams. However, the results at the highest maturity need to be operationalized and examine to verify the interrelationships between quality, cycle time and effort in real projects. Further research could explore the practicality of the processes and its implementation, enabling a greater reduction in the project cycle time. These results hopefully will provide useful insights for planning and project management methodology in assessing the return on investment from quality improvement in the development process.

References

[1] Ivo, G., Pedro, M., & Tiago, G. An overview on the static code analysis approach in software sevelopment. Faculty of Engineering, University of Porto, Portugal.

[2] Rover, D. T., & Waheed, A. (1998). Software tools for complex distributed systems: Towards integrated tool environments. *Concurrency*, *6(2)*, 40-54.

[3] Deissenboeck, F., & Juergens, E. (2008). Tool support for continuous quality control. *IEEE Software*, *25(5)*, 60-67.

[4] Vladimir, A., Jasbir, D., & Thomas, M. (2011). Implementing quality gates throughout the enterprise IT production process. *Journal of Information Technology Management Volume XXII*.

[5] Code quality governance. Retrieved, from http://www.codeexcellence.com/2012/04/code-quality-governance-what-is-it-and-why-should-you-care/

[6] *Controlling Risk Through Software Code Governance*. Coverity White Paper.

[7] Nanette, B., Yuanfang, C., & Yeupu, G. (2010). *Managing technical debt in software-reliant systems*. Santa Fe, New Mexico, USA.

[8] K, Philippe., L, Robert., & O, Ipek. (2012). Technical debt: From metaphor to theory and practice. *IEEE Software*, *29(6)*, 18-21.

[9] Release gate. Retrieved, from https://developer.ibm.com/urbancode/products/urbancode-release/features/release-gate/

[10] Quality gates approvals. Retrieved, from https://developer.ibm.com/urbancode/products/urbancode-deploy/features/quality-gates-approvals/

[11] Krishnan, H. D. E., & Slaughter, S. A. (2000). Effects of process maturity on quality, cycle time and effort in software product development. *Management Science*, *46(4)*, 451-466.

[12] Zheng, J., & Laurie, W. (2006). On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, *32(4)*.

[13] Subramanyam, V., Sambuddha, D., Priya, K., & Rituparna, G. (2004). An integrated approach to software process improvement at wipro technologies: Veloci-Q. Wipro Technologies and Carnegie Mellon University, US.

[14] Barry, J., & Slaughter, K. F. A. (2007). How software process automation affects software evolution: a longitudinal empirical analysis. *Journal Softw*, 1-31.

[15] Jamaiah, Y., Aziz, D., & Razak, A. H. (2007). Software certification model and experience of its use. *Proceedings of the Regional Conference on Computational Science and Technology*.

[16] Ibrahim, Z., & Naseb A. (2007). Total quality management in service industry: A study on the Islamic banking in Malaysia and yemen. *Proceedings of the International Conference on Business and Economic Research*.

Fariha Motherudin recieved her IT degree majoring in information system engineering from Multimedia University, Cyberjaya Malaysia in 2003.

She is currently a senior software quality engineer in MIMOS Bhd, Kuala Lumpur, Malaysia who has vast ex perience in software development processes supporting more than 20 projects. Her technical background developing web applications for several clients and a product executive of major broadband product in Malaysia really helped her in understanding the overall company's operation. She had submitted conference papers to SEPG, MySEC and other conferences. Her technical interests include governance, code level quality, continuous integration, review inspection and process automation.

Noor Ezleen Md. Moksen is originally from Malaysia. She received her master's degree in computer science majoring in real time software engineering from University Technology of Malaysia in 2004.

She is a senior software quality engineer in MIMOS Bhd, Kuala Lumpur, Malaysia who has 16 years experience in software development processes. She came from a quality assurance background and currently part of a process improvement team in her organization. She had submitted conference papers to SEA, ASWEC, and other conferences. Her technical interests include measurement and analysis, process model and project management.