International Journal of Engineering and Technical Research (IJETR)
ISSN: 2321-0869, Volume-3, Issue-4, April 2015

Software Quality

Mrs. Pinki

Abstract— **Software quality is one of the most pressing concerns for nearly all software developing companies. At the same time, software companies also seek to shorten their release cycles to meet market demands while maintaining their product quality. Identifying problematic code areas becomes more and more important. Defect prediction models became popular in recent years and many different code and process metrics have been studied. There has been minimal effort relating test executions during development with defect likelihood. This is surprising as test executions capture the stability and quality of a program during the development process. This paper presents an exploratory study investigating what is software quality, why is it important and how it can be maintained.**

***Index Terms*—Software quality, RAM, code characteristics.**

I. Software Quality

The quality of software is assessed by a number of variables. These variables can be divided into external and internal quality criteria. External quality is what a user experiences when running the software in its operational mode. Internal quality refers to aspects that are code-dependent, and that are not visible to the end-user. External quality is critical to the user, while internal quality is meaningful to the developer only. By definition the internal quality (code characteristics) is a concern to the developer only, while all the external quality aspects (coming from using the software) are critical to the end user. Hoawever the developer has also interests in performances (speed, space, network usage) and determinism, because they make testing the software easier. Developers treat ease-of-use, back-compatibility, security, and power consumption as requirements.

It is important to consider how difficult it is to measure each of these criteria. It can be difficult because there is no simple variable to look at, or because the measurement process is costly, or because it requires a complex infrastructure. For instance, speed has an objective measurement that is easy to measure. Power consumption has a simple measurement, but it is complex to measure. Security is difficult and costly to estimate.

Features: This is the very reason for the software to be written: to provide a service. By feature we really mean the output produced by the software –e.g., a numerical result, a string, a screen shot, a web page, an audio, etc–, regardless of the performances (speed, memory).

Speed: How quickly does the application provide the service? The user experiences the actual time elapsed between the moment she request the service, and the moment the service is delivered. The real elapsed time, or wall time, is the sum of the CPU time, system time, and network latency. Thus the developer should not only focus on the CPU time (how much time the CPU actually spends on executing the program). The CPU time can easily be overshadowed by disk access (a write on the disk is very costly), swapping (due to an excessive virtual memory size), or time spent by the network (latency issue, or too many round trips).

Space: How much RAM and disk space is taken by the application? The aggregate numbers are important –peak memory, virtual memory size, etc. But even more so, how often do we move data that triggers a CPU cache miss or a disk write, has a dominant impact on the speed of the application. A mediocre data design can lead to very poor performances.

Network usage: It is a matter of bandwidth and latency. Mismanaging sockets and channels can lead to unnecessary extra time spent in opening and closing sockets, handshakes, and round trips. As for memory, caching techniques can be used to reduce consuming network resources.

Stability: How often does one need to patch the software to fix problems? For the user, this is an inconvenience. For the developer, it means that the code is fragile and might benefit from better testing or partial rewrite.

Robustness: How often does the application stale, freeze, or crash? How tolerant is it to extreme conditions –limited CPU and memory/disk/network resources, corner cases, system failure or unresponsive 3rd-party resources? This aspect is strongly related to testability and coverage.

Ease-of-use: It can be a very subjective factor, hard to quantify. It includes user documentation, clarity of the error message, management of exceptions, and recovery after failure.

Determinism: Also known as repeatability: does the program produce the very same result given the same input? There are many reasons for which a program can exhibit a non-repeatable behavior. A non-repeatable behavior is confusing and frustrating for the user. This also makes the program very difficult to test and debug. Repeatability is strongly dependent on a good data model design.

Back-compatibility: Can a new version of the application be used with an older version's data? It is essential to the user, because a new version should not require a costly migration of the existing data.

Security: Who is authorized to access the data? Can the data processed by the application be compromised? This is a crucial aspect of many applications, and it is getting more and more difficult to assess with the dissemination of mobile and web-based software.

Power consumption: It is increasingly important with mobile applications, as a program may have to consider how it manages the device's power producers and consumers (battery, cores, wireless, screen, audio), and not to rely entirely on the operating system.

Test coverage: What is the proportion of code that is executed by some unit or regression test? This is measured by the number of lines, number of functions, and number of control branches that are exercised by the tests. Usually one

Manuscript received March 30, 2015.

Mrs. Pinki is pursuing her M.Tech Degree in Computer Science & Engineering from Sat Kabir Institute Of Technology & Management Bahadurgarh.

expects coverage of at least 85% for any moderately complex application. In practice reaching high coverage can be achieved only if testability is high, which has deep implication on the architecture and development methodology.

Testability: An often overlooked or simply ignored aspect of code development, testability is the ability to trigger any specific line of code or branching condition. Highly testable code requires a discipline of architecture and development that is difficult to find. It very costly to fix poorly testable software, as this requires major redesign. This justifies major investment in software architecture, design, and development methodologies.

Portability: Can the application run on 32 and 64 bits machines? Should it run on a mobile phone? Does it run on multiple OS (e.g., Windows, Linux, Mac OS-X, Solaris, iOS, Android, RIM)? Does it run smoothly on all web browsers (IE, Firefox, Chrome, Safari, Opera)?

Thread-safeness: Is a specific component thread-safe? Can two threads collide on non-atomic operations? Can the application get into a deadlock? As concurrency is still mostly the result of a manual process (there no compiler that automatically parallelizes the code), these questions are critical to ensure the good functioning of a program, as well as its performance –it is not rare to see the a program running *slower* when two many threads are available, as the cost of synchronization can become dominant.

Conciseness: Also known as compactness. Is there any dead code, or duplicated code? Is the code shared and factorized properly? A compact code usually means faster compilation and smaller binary size. Also compactness naturally leads to fewer bugs, because the number of bugs is historically constant w.r.t. code size.

Maintainability: How easy it is to debug the code? How fast is it to provide a fix? How quickly can a new developer understand the code? Maintainability is a very important aspect, quite difficult to quantify. Maintainability is increased with good testability and flexible (abstract) design.

Documentation: This is a pretty subjective topic. Some people claim that a separate documentation written in plain English is necessary. Some others state that at least 30% of the code should be comments. Some finally argue that the code itself is the best documentation –the names of the types, classes, functions and arguments, together with plenty of assertions.

Legibility: Also known as readability. This is another subjective topic. It is about how easy it is to read the code. Guidelines are established to unify the style of the code, so that a developer can easily read code written by another developer. Code guidelines abound, and they go from a small set of directives, to a full set of rules that specify every syntactical aspect of the language.

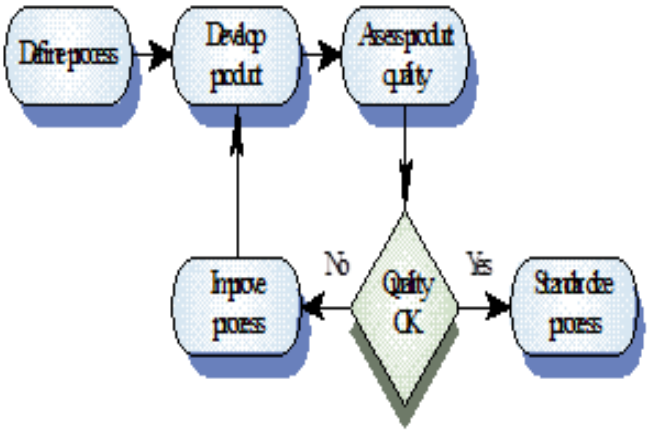
Scalability: How easy it is to extend a feature? Or to add a new one? Or to add extra cores, or increase the size of the cluster the application runs on? Again, this is all about software architecture and anticipating future needs.

Software quality is the result of the user experience. But software quality should not and cannot be a reactive action to external defects. Software quality is built from the ground up, with design and development methodologies, and with a special focus on testability, coverage, and flexibility.

Need of Software Quality: If you don't focus on product quality then:

• You tend to produce components with more (hidden) defects, so
• You have to spend more time fixing these (late), so
• You have little time for anything else, so
• You produce poor quality software even though you put huge amounts of effort into defect checking.

Thus quality is something that has to be considered throughout the product lifecycle; it cannot be added in later.



Thus it makes sense to focus on improving component quality before testing, to catch difficult defects early.

II. Software Quality Management

Software Quality Management simply stated comprises of processes that ensure that the Software Project would reach its goals. In other words the Software Project would meet the clients expectations.

The key processes of Software Quality Management fall into the following three categories:

1) Quality Planning
2) Quality Assurance
3) Quality Control

III. Quality Planning

Quality Planning is the most important step in Software Quality Management. Proper planning ensures that the remaining Quality processes make sense and achieve the desired results. The starting point for the Planning process is the standards followed by the Organization. This is expressed in the Quality Policy and Documentation defining the Organization-wide standards. Sometimes additional industry standards relevant to the Software Project may be referred to as needed. Using these as inputs the Standards for the specific project are decided. The Scope of the effort is also clearly defined.

IV. Quality Assurance

The Input to the Quality Assurance Processes is the Quality Plan created during Planning.
Quality Audits and various other techniques are used to evaluate the performance of the project. This helps us to ensure that the Project is following the Quality Management Plan.

The tools and techniques used in the Planning Process such as Design of Experiments, Cause and Effect Diagrams may also be used here, as required.

V. Quality Control

Following are the inputs to the Quality Control Process:

- Quality Management Plan.
- Quality Standards defined for the Project
- Actual Observations and Measurements of the Work done or in Progress

The Quality Control Processes use various tools to study the Work done. If the Work done is found unsatisfactory it may be sent back to the development team for fixes. Changes to the Development process may be done if necessary.

In a typical Software Development Life Cycle the following steps are necessary for Quality Management:

1) Document the Requirements
2) Define and Document Quality Standards
3) Define and Document the Scope of Work
4) Document the Software Created and dependencies
5) Define and Document the Quality Management Plan
6) Define and Document the Test Strategy
7) Create and Document the Test Cases
8) Execute Test Cases and (log) Document the Results
9) Fix Defects and document the fixes
10) Quality Assurance audits the Documents and Test Logs

VI. Conclusion

A fixed software quality model is often helpful for considering an overall understanding of software quality. In practice, the relative importance of particular software characteristics typically depends upon software domain, product type, and intended usage. Thus, software characteristics should be defined for, and used to guide the development of, each product. Quality function deployment provides a process for developing products based on characteristics derived from user needs. This paper includes the importance of software quality, the attributes of software quality, what is the need of software quality and what is software quality management.

VII. REFERENCES

[1] V. Basili, Briand, L., Melo, W., "A Validation of Object Oriented Design Metrics as Quality Indicators", IEEE Transactions on Software Engineering, 22(10), pp. 751- 761, 1996.
[2] V. Basili, G. Caldiera, and D. H. Rombach, "The Goal Question Metric Paradigm," in Encyclopedia of Software Engineering, vol. 2: John Wiley and Sons Inc., 1994, pp. 528-532.
[3] V. Basili, Shull, F.,Lanubile, F., "Building Knowledge through Families of Experiments", IEEE Transactions on Software Engineering, 25(4), pp. 456-473, 1999.
[4] S. Biyani, Santhanam, P., "Exploring defect data from development and customer usage on software modules over multiple releases", Proceedings of International Symposium on Software Reliability Engineering, pp.316-320, 1998.
[5] L. C. Briand, Wuest, J., Ikonomovski, S., Lounis, H.,"Investigating quality factors in object-oriented designs: an industrial case study", Proceedings of International Conference on Software Engineering, pp. 345-354, 1999.
[6] Roger S. Pressman, Software Engineering – A Practitioners Approach, Fifth Edition, McGraw –Hill, 2005.
[7] S. R. Chidamber, Kemerer, C.F., "A Metrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering, 20(6), pp. 476-493, 1994.
[8] Ian Sommerville, Software Engineering, Sixth Edition, Perason Education,2001.
[9] T. DeMarco and T. Lister, Peopleware. New York: Dorset House Publishers, 1977.
[10] T. M. Khoshgoftaar, Szabo, R.M., "Improving Code Churn Predictions During the System Test and Maintenance Phases", Proceedings of IEEE International Conference on Software Maintenance, pp.58-67, 1994.
[11] D. G. Kleinbaum, Kupper, L.L., Muller, K.E., Applied Regression Analysis and Other Multivariable Methods. Boston: PWS-KENT Publishing Company, 1987.
[12] T. J. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, 2(4), pp. 308- 320, 1976.
[13] A. Mockus, Fielding, R.T., Herbsleb, J., "Two case studies of open source software development: Apache and Mozilla", ACM Transactions on Software Engineering and Methodology, 11(3), pp. 309 - 346, 2002.

Mrs. Pinki is pursuing her M.Tech Degree in Computer Science & Engineering from Sat Kabir Institute Of Technology & Management Bahadurgarh (Affiliated to Maharishi Dayanand University Of Science and Technology, Rohtak, Haryana (INDIA)). She received her B.Tech. in Computer Science & Engineering from B.P.R . College of Engineering, Gohana (Sonipat) (Affiliated to M.D.U. Rohtak) in 2009. Her research interest includes Software Engineering Concepts and Computer Networks