TEST CODE QUALITY WITH ISSUE HANDLING PERFORMANCE

[*1] Ms. Harini G., [*2] Mrs. Shoba S.A.,

[*1] M.Phil Research Scholar, PG & Research Department of Computer Science & Information Technology Arcot Sri Mahalakshmi Women's College, Vellore, Tamil Nadu, India

[*2]Assistant Professor, HOD of PG & Research Department of Computer Science & Information Technology Arcot Sri Mahalakshmi Women's College, Vellore, Tamil Nadu, India

---***---

INTRODUCTION: Testability is a major quality factor for producing high quality software. Lack of testability contributes to increased test and maintenance effort. The IEEE Standard Glossary defines testability as the degree to which a system or component makes possible the establishment of test Criteria and performance of tests to conclude whether those criteria have been ISO defines it in a parallel way: an attributes of software that bear on the effort needed to validate the software product. The most well-known definition of testability is easiness of performing testing. The insight provided by software testability is significant for the extent of development life cycle and quality promise. Design-for-testability is a very important issue in software engineering.

Testability is one of the most important factors determining the time and effort required to test software system. A lower degree of testability outcome means increased test effort. It is essential in the case of Object Oriented designs where control flows are normally not hierarchical; it is costly to redesign a system during implementation or maintenance. It has been concluded that Flexibility and Modifiability are the two most important factor affecting software testability measurement at design phase. Taking into consideration the significance of their involvement, in this paper we have proposed a model to measure software flexibility at design phase.

Software Quality is to calculate a process of method and components of a system meeting the necessities that are already specified. We can also say that in which it can assemble customers or users necessities also. Relatively a single factor, quality in software is best viewed as a tradeoff between a set of different goals. Explicit attention to uniqueness of software quality can lead to important savings in software life-cycle costs.

Distinctiveness of good quality software includes the Understand ability, Completeness, Conciseness, Portability, Consistency, Maintainability, Testability, Usability and Reliability.

Software metric is one of the significant aspects of software engineering acts as an indicator for software attribute. It plays a significant role in understanding the vital concepts in the field of software engineering Software Metrics can be defined by measuring.

Software metrics explains the activities connected with measurements in software engineering. The metrics are practical to software development process and the product so as to get the significant information. Software metrics is classified into two types.

- Static metric
- Dynamic metric

Software Testing and debugging is concerned with the discovery of defects regarding the functionality and reliability as defined in a specification or unit test case in static and dynamic environments. Software product metrics are used in software analysis to measure the complexity, cohesion, coupling, or other characteristics of the software product.

II. Related work

In broadest terms the properties associated with structural forms that impact the quality of software involve two fundamental things: *correctness* and *style.*
This is purely an empirical heuristic decision. However it is not hard to justify to most people that a violation of a computability property is likely to have a much more significant impact on correctness than violation of a consistency property.

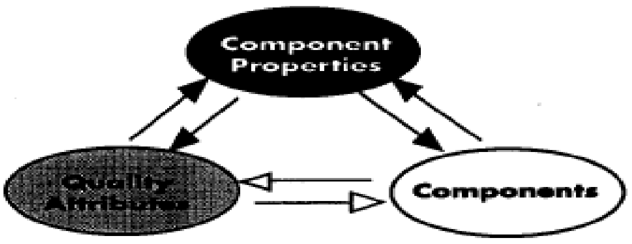

Fig. 1.1 Generic Quality Model

The structural properties focus upon the way individual statements and statement components are implemented and the way statements and statement blocks are composed, related to one another and utilized. They enforce the requirements of structured programming and demand that there should be no logical, computational, representational and declarative redundancy or inefficiency of any form either in individual statements or in sequences or in components of statements. The modularity properties employed largely address the high-level design issues associated with modules and how they interface with the rest of a system.

2.1 Test Code Quality

The main role of test adequacy criteria is to assist software testers to monitor the quality of software in a better way by ensuring that sufficient testing is performed. In addition, redundant and unnecessary tests are avoided, thus contributing to controlling the cost of testing; follow the classification of test adequacy criteria. Program-based test adequacy criteria can be subdivided into categories for structural testing, fault-based testing and error-based testing.

Structural Testing Adequacy Criteria

This category consists of test criteria that focus on measuring the coverage of the test suite upon the structural elements of the program. These criteria can be further split between control-flow criteria and data-flow criteria. They are mostly based on analysis of the flow graph model of program structure.

Error-Based Testing Adequacy Criteria

This category of test criteria focuses on measuring to what extent the error-prone points of a program. To identify error-prone points, a domain analysis of a program’s input space is necessary. Unfortunately, the application of error-based testing is limited when the complexity of the input space is high or when the input space is non-numerical.

2.2 Issue Handling

Issue Tracking Systems and the Life-Cycle of an Issue

Software systems used to track defects as well as enhancements or other types of issues, such as patches or tasks. It is commonly used and they enable developers to organise the issues of their projects. Issues that are reported follow a specific life-cycle.

Defect Resolution Time

Defect resolution time is an indicator of the time that is needed to resolve a defect. An arguably straightforward measurement of the defect resolution time is to measure the interval between the moment when the defect was assigned to a developer and the moment it was marked as resolved.

Throughput and Productivity

Throughput and productivity measures the level of issues and thus comprise both defects and enhancements. Both measures capture the number of issues that are resolved in a certain time period, corrected for respectively the size of the system and the number of developers working on the system. Throughput measures the total productivity of a team working on a system in terms of issue resolution

Throughput = # Resolved Issues per Month / KLOC

The Productivity is defined as follows:

Productivity = #resolved Issues per Month / # Developers

III. Previous Implementations

Regression test selection (i.e., selecting a subset of a given regression test suite) is a problem that has been studied intensely over the last decade. However, with the increasing popularity of developer tests as the driver of the test process more fine-grained solutions are in order. In this paper author investigate how method-level changes in the base-code can serve as a reliable indicator for identifying which tests need to be rerun.

Unit and integration tests can be invaluable during software maintenance as they help to understand pieces of code they help with quality assurance and they build up confidence amongst developers. Unfortunately then, previous research has shown that unit tests do not always co-evolve nicely with the production code, thus leaving the software vulnerable. This paper presents TestNForce, a tool that helps developers to identify the unit tests that need to be altered and executed after a code change, thereby reducing the effort needed to keep the unit tests in sync with the changes to the production code.

Designing automated tests is a challenging task. One important concern is how to design test fixtures, i.e. code that initializes and configures the system under test so that it is in an appropriate state for running particular automated tests. Test designers may have to choose between writing in-line fixture code for each test or refactor fixture code so that it can be reused for other tests. Deciding on which approach to use is a balancing act, often trading off maintenance overhead with slow test execution. Additionally over time, test code quality can erode and test smells can develop such as the occurrence of overly general fixtures obscure in-line code and dead fields. That test smells related to fixture set-up occurs in industrial projects. Author present a static analysis technique to identify fixture related test smells.

Software metrics have been proposed as instruments not only to guide individual developers in their coding tasks but also to obtain high-level quality indicators for entire software systems. Such system-level indicators are intended to enable meaningful comparisons among systems or to serve as triggers for a deeper analysis. To resolve such limitations, a two stage rating approach has been proposed where (i) measurement values are compared to thresholds to summarize them into risk profiles and (ii) risk profiles are mapped to ratings.

IV. SYSTEM IMPLEMETNATION

The mapping of metrics to the sub-characteristics is done, with the note that the adjusted SIG (Software Improvement Group) quality model combines duplication, unit size, unit complexity and unit dependency into a maintainability rating. The aggregation of the properties per sub-characteristic is performed by obtaining the mean. For maintainability, this is done separately in the adjusted maintainability model. The aggregation of the sub-characteristics into a final, overall rating for test code quality is done differently. The overall assessment of test code quality requires that all three of the sub-characteristics are of high quality.

Test Code Quality = Completeness + Effectiveness + Maintainace

4.1 Properties

Code coverage

Code coverage is the most frequently used metric for test code quality assessment and there exist many tools for dynamic code coverage estimation. The fore mentioned tools use a dynamic analysis approach to estimate code coverage.

Assertions-McCabe Ratio

The Assertions-McCabe ratio metric indicates the ratio between the number of the actual points of testing in the test code and of the decision points in the production code.

Assertion – McCabe Ratio = #assertion / Cyclomatic Complexity

Cyclomatic Complexity	**Risk Categoery**
1 to 10	Low
11 to 20	Moderate
21 to 50	High
>50	Very High

Table 1.1 McCabe's Cyclomatic Table

Assertion Density

Assertion density aims at measuring the ability of the test code to detect defects in the parts of the production code that it covers.

Assertion Density = #Assertions / LOC

Directness

When each unit is tested individually by the test code, a broken test that corresponds to a single unit immediately pinpoints the defect. Directness measures the extent to which the production code is covered directly.

4.2 The Evaluation Process of Software

A standard evaluation procedure has been defined in which the quality model is applied to software product.

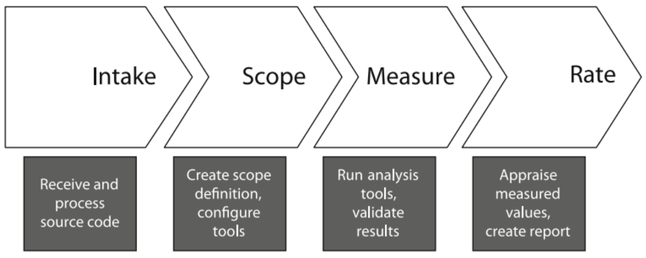
Intake
Scope
Measure
Rate
Receive and process source code
Create scope definition, configure tools
Run analysis tools, validate results
Appraise measured values, create report

Fig. 1.2 Evaluation Procedure

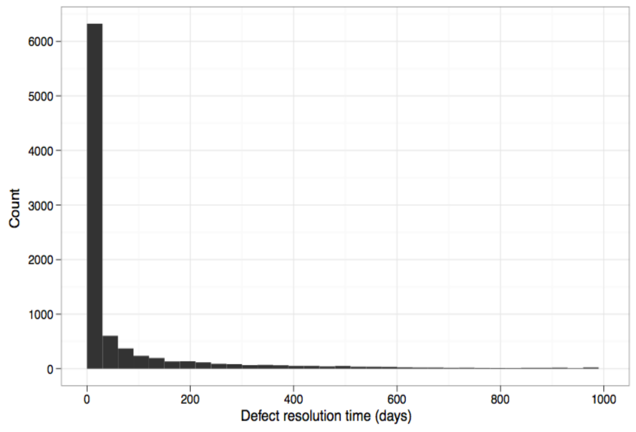

Fig. 1.4 Defect Resolution Time

4.3 Test Code Quality Model

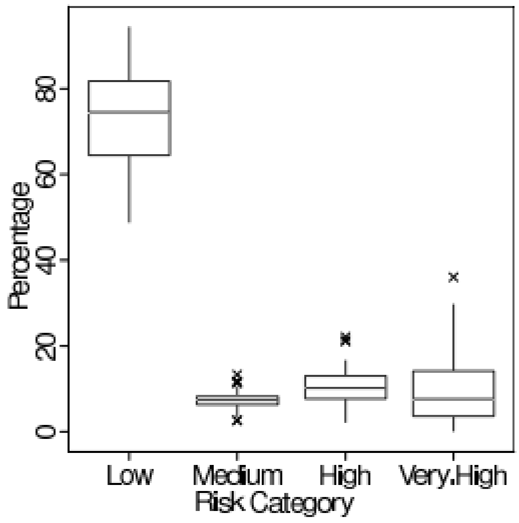

Fig. 1.3 Quality Profile Variability

Here the X-axis represents the four risk categories, and the Y-axis represents the percentage of volume (lines of code) of each system per risk category.

4.4 Defect Resolution Speed Rating

The dependent variable 1 is the resolution time of defects in a system, which is measured by calculating a rating that reflects the defect resolution speed.

EVALUATION RESULT:

Project	Data for Defect Resolution Speed			Data for Throughput and Productivity		
	Issues	Defects	Enhancements	Issues	Defects	Enhancements
Java Application	2275	1680	595	1944	1467	477
Apache Ivy	331	228	103	467	309	158
Apache Lucene	2222	1547	675	4092	3274	818
Apache Tomcat	296	268	27	275	244	31

Table 1.2 Issues Per System

	Defect Resolution	
	ρ	p-value
Code Coverage	0.28	0.013
Assertion McCabe Ratio	0.01	0.48
Assertion Density	0.02	0.427
Directness	0.08	0.26

Table 1.3 Results for Throughput

	Productivity					
	Defects		Enhancements		Combined	
	P	p-value	ρ	p-value	ρ	p-value
Code Coverage	0.51	0	0.43	0	0.49	0
Assertion McCabe Ratio	0.48	0	0.56	0	0.53	0
Assertion Density	0.26	0.026	0.32	0.008	0.33	0.007
Directness	0.32	0.009	0.43	0	0.36	0.004

Table 1.4 Results for Productivity

The java application is accepted, because the highest McCabe Ratio is achieved in both throughputs and productivity. Hence, that the java application has the best quality software.

CONCLUSION

Developer testing is an important part of software development. Automated tests enable early detection of defects in software, and facilitate the comprehension of the system. The three main aspects of test code quality that we identified are: completeness, effectiveness and maintainability. Completeness concerns the complete coverage of the production code by the tests. Effectiveness indicates the ability of the test code to detect defects and locate their causes. Maintainability reflects the ability of the test code to be adjusted to changes of the production code, and the extent to which test code can serve as documentation. Suitable metrics were chosen based on literature and their applicability. Code coverage and assertions-McCabe ratio are used to assess completeness. Assertion density and directness are indicators of effectiveness. Test code quality is mapped with issue handling performance techniques. The test code quality is measured by using these metrics.

FUTURE WORK

The current test code quality model is solely based on source code measures. It might be interesting to extend the model with historical information that would bring additional insight as to the number of previous bugs (defects that were not caught by the test code). To assess the relation between test code quality and issue handling performance we used three issue handling indicators. However, other indicators reflect different aspects of issue handling, e.g., the percentage of reopened issues could provide an indication of issue resolution efficiency. Future research that includes additional indicators will contribute to the knowledge of which aspects of issue handling are related to test code quality in particular.

References:

1. T. L. Alves and J. Visser. *"Static estimation of test coverage".* In Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, IEEE Computer Society, 2009.
2. T. L. Alves, C. Ypma, and J. Visser. *"Deriving metric thresholds from benchmark data".* In Proceedings of the 2010 IEEE International Conference on Software Maintenance, IEEE Computer Society, 2010.
3. J. An and J. Zhu. *"Software reliability modeling with integrated test coverage".* In Proceedings of the 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement, IEEE Computer Society, 2010.
4. R. Baggen, J. Correia, K. Schill, and J. Visser. *"Standardized code quality benchmarking for improving software maintainability".* Software Quality Journal, 2011.
5. A. Bertolino. *"Software testing research: Achievements, challenges, dreams".* In 2007 Future of Software Engineering, IEEE Computer Society, 2007.
6. D. Bijlsma. *"Indicators of Issue Handling Efficiency and their Relation to Software Maintainability".* Msc thesis, University of Amsterdam, 2010.
7. B. Luijten. "*The Influence of Software Maintainability on Issue Handling".* Master's thesis, Delft University of Technology, 2009.
8. B. Luijten and J. Visser. *"Faster defect resolution with higher technical quality of software".* In 4th International Workshop on Software Quality and Maintainability (SQM 2010), 2010.
9. N. Nagappan, L. Williams, M. Vouk, and J. Osborne. *"Early estimation of software quality using in-process testing metrics: a controlled case study".* SIGSOFT Softw. Eng., May 2005.
10. F. Oppedijk. *"Comparison of the SIG Maintainability Model and the Maintainability Index".* Master's thesis, University of Amsterdam, 2008.