Improving Software through Metrics while Providing Cradle to Grave Support

Jennifer Walters, Northrop Grumman
Kevin MacG. Adams, Ph.D., NCSOSE

Abstract. Metrics are beneficial to an organization that supports a product from inception through product retirement and disposal. Quality metrics have a critical role in this type of environment because they span both the development and operations and maintenance phases of the software life cycle, and there is a relationship between the internal quality metrics collected during development and the external quality metrics collected once the product is deployed. The key finding is that internal metrics can be collected early in the software development phase to predict the support required during the operations and maintenance phase; likewise, external metrics can be collected to drive software development process improvements. Finally, analyzing the relationships between the two can drive overall process improvements for the entire software lifecycle.

Introduction

Software development is "the specification, construction, testing and delivery of a new application or of a discrete addition to an existing application" [1] while a maintenance project is "a software development project described as maintenance to correct errors in an original requirements specification, to adapt a system to a new environment, or to enhance a system" [1]. Often times these two processes are supported by two different organizations with two different goals. One team will focus on building the initial application and their primary concern is building a product that fulfills the requirements within both cost and schedule constraints. The second team has the responsibility of supporting the product during the remainder of the lifetime and has a primary concern of maintainability.

As a result, the maintenance team is at the mercy of the software design and processes imposed by the software development team. While the resulting product might meet all the user requirements and appear to be a superb product, it is likely that the software development team did not build the initial product with maintenance in mind [2]. This results in a product that is more difficult and costly to maintain.

One Organization Supporting Entire Lifecycle

When the same organization supports both the development phase and the operations and maintenance phase, however, there are some opportunities to create a synergy between development and maintenance efforts. The focus can shift from individual phases to the overall software lifecycle. By transitioning focus, team members can collect measurements early in the development phase that can help predict issues in maintenance. Likewise, maintenance related metrics can be analyzed to provide guidance for improving development processes. Cause and effect analysis is a very powerful technique in this situation.

Role of Metrics

It is important to understand the role metrics play in the overall software lifecycle. First, the metrics of concern in this paper are quality attributes because maintenance efforts are highly dependent on the overall quality of the software [3]. In addition, quality attributes span both pre-delivery and post-delivery phases of the lifecycle and are therefore specifically relevant when a single organization supports the software over the entire lifecycle. Second, quality attributes are categorized as either internal or external [4]. "Internal quality attributes are those that can be directly measured purely on the basis of product features such as size, length, or complexity" [5]. External metrics are measurements that are dependent on how the software interacts with its environment and can therefore only be collected after the product has been deployed and operated during the maintenance and operations phase of the software lifecycle [5]. The remainder of this discussion involves the relationship between internal and external metrics.

Internal Metrics

As previously mentioned, internal quality attributes are measurements based on characteristics of the product itself. Size, length, and complexity are the key attributes collected. Research has shown that there is a correlation between internal quality attributes and external quality of the product [5-8]. By measuring internal quality that can be accomplished early during the software development process, it is possible to predict product maintainability and thus the effort required to support product maintenance.

There are many metric options available for evaluating internal software design quality. A few of the most commonly used suites include: Chidamber and Kemerer (CK) Metrics [9], Robert C. Martin Metric Suite [10], and McCabe's Metric Suite [11,12]. Table 1 provides a brief overview of the metric suites. The next section will discuss CK Metrics in more detail.

Chidamber and Kemerer Metrics

Chidamber and Kemerer (CK) metrics can "assist users in understanding object oriented design complexity and in forecasting external software qualities for example software defects, testing, and maintenance effort" [13]. Numerous research studies have validated CK metrics as a method for predicting maintainability [8, 13-16]. The suite includes six metrics: (1) weighted methods per class (WMC), (2) depth of inheritance tree (DIT), (3) number of children (NOC), (4) coupling between objects/classes (CBO), (5) response for a class (RFC), and (6) lack of cohesion in method (LCOM). Each of the six measurements in the CK suite quantify different quality attributes which relate to maintainability qualities; however, the last metric discussed in the next section, LCOM, has been shown to have the greatest impact on the total number of defects [8].

Weighted Methods per Class

Weighted Methods per Class (WMC) is a complexity measurement. However, Chidamber and Kemerer did not propose

a definition or method for measuring complexity. "If methods complexities are considered to be unity, the WMC metric turns in to the number of methods in a class" [17]. Whether it is generalized to a count of methods or is more specialized through the use of a complexity algorithm, a higher WMC indicates a class that is more difficult to understand and modify.

Depth of Inheritance Tree

Depth of Inheritance Tree (DIT) is object oriented (OO) specific as it measures the OO characteristic inheritance. Inheritance is "a semantic notion by which the responsibilities (properties and constraints) of a subclass are considered to include the responsibilities of a superclass, in addition to its own, specifically declared responsibilities" [1]. It is often described as an isa relationship. For example, a cat isa mammal. This can be extended to include a mammal isa animal. DIT measures the hierarchy of inheritance. In this example, there is a hierarchy of three: cat > mammal > animal. The deeper a class is in the inheritance tree, the more difficult it is to comprehend and predict the behavior of the class [18].

Number of Children

Number of Children (NOC) is similar to DIT as it is related to inheritance. It is the count of immediate sub-classes in the class hierarchy [13]. NOC takes a more horizontal approach to inheritance. Instead of walking down the inheritance tree, it counts the number of classes inheriting methods from the parent class. Extending the previous example, a reptile is also an animal. Now both mammal and reptile classes inherit the members from animal. So, if the animal base class is modified there is potential impact to both sub-classes. High NOC measurements require more impact analysis.

Coupling Between Objects Classes

As the name suggests, Coupling Between Objects Classes (CBO) measures coupling. Coupling is "the manner and degree of interdependence between software modules" [1]. A class is considered coupled with another class if its members are used by another class. Coupling makes it more difficult to isolate units of code for testing. The interdependence also makes code comprehension more difficult and increases the need for impact analysis. It is "highly connected to portability, maintainability, and re-usability" [17].

Response for a Class

Response for a Class (RFC) is similar to CBO but this measure also takes inheritance into count. It is the number of methods that can be executed as a response to a message received by class objects [13,17]. The count includes methods in the same class, methods accessible within the class hierarchy, and methods accessible in other classes. Source code with a high RFC count is complex and can be very difficult to trace potential code paths for testing and comprehension.

Metric Suite	Description	Measurements
CK Metrics	CK metrics designed specifically for object-oriented software [9]. Commonly utilized to predict fault-proneness [8] and included in static code analysis tools to provide automation opportunities.	• Weighted Methods Per Class (WMC) • Depth of Inheritance Tree (DIT) • Number of Immediate Subclasses (NOC) • Coupling between Objects Classes (CBO) • Response for a Class (RFC) • Lack of Cohesion in Method (LCOM)
Robert C. Martin Metric Suite	This suite of metrics focuses on interdependence between packages, or cohesion [10]. It is also designed for object-oriented software.	• Afferent Coupling • Efferent Coupling • Instability • Abstractness • Normalized Distance from Main Sequence
McCabe's Metric Suite	McCabe is most commonly associated with the concept of cyclomatic complexity. The metric was introduced as a way to quantify design decisions to indicate how difficult it is to test and maintain the method [11, 12]. Also commonly found in static analysis tools.	• McCabe's Complexity • Method Lines of Code • Total Lines of Code • Nested Block Depth

Table 1: Metric Suites for Measuring Internal Quality Attributes

Lack of Cohesion in Method

The final metric in the CK suite is Lack of Cohesion in Method (LCOM). Since cohesion is considered a positive characteristic in object-oriented code, this measurement considers the lack of cohesion. "A highly cohesive module should be independent" [13] of other modules and improve reusability. Lack of cohesion on the other hand signifies poor design that generates more complex code that is difficult to maintain. It is an indicator of potential redesign opportunities to create smaller more cohesive classes [13,17]. LCOM "has a significant effect on the total number of defects… and software development companies should concentrate on [it] to control… design defects" [8].

External Metrics

While internal metrics provide a wealth of interesting information, the measures themselves are not very helpful on their own. "For these early indicators to be meaningful, they must be related (in a statistically significant and stable way) to the field quality/reliability of the product" [7]. The field quality attributes can only be measured while the product is in the testing or operations and maintenance phase of the software lifecycle (ISO/IEC, 2002) and are known as external metrics. Figure 1 displays the relation-

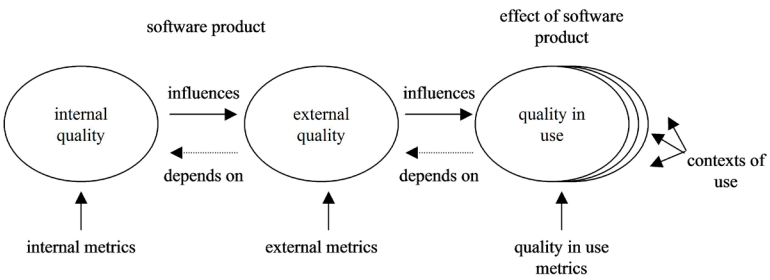


Figure 1: Relationships between types of metrics [19, p.4]

While much emphasis is placed on the development phase, most software products do eventually reach the operations and maintenance phase where they must then be maintained until the system is no longer needed or is replaced by a new product. It stands to reason, then, that software should be developed with maintenance in mind. But this is not typically the case because the software development team is focused on quickly creating a product to meet the needs of the customer at a cost and schedule the customer is willing to accept. The team is likely not considering how the product will be maintained after it is in operation.

ship between the internal metrics already discussed and external metrics which include functionality, usability, reliability, maintainability, portability, and efficiency [19].

As shown, internal metrics influence external quality and external quality depends on internal metrics. With this type of relationship it is possible to predict external quality as well as maintenance support needs of the final product by collecting internal metrics which are available earlier in the software lifecycle [7,8,13,19].

Maintainability

While all of the external metrics are important to a maintenance organization, maintainability is critical when planning and staffing maintenance activities and will be the focus of the external metric discussion that follows. Maintainability is "the ease with which software system or component can be modified to change or add capabilities, correct faults or defects, improve performance or other attributes, or adapt to a changed environment" [1]. The four primary maintainability metrics are analyzability, changeability, stability, and testability.

Analyzability

Analyzability describes the ability to trace and comprehend the intent of the existing software code. The maintenance organization must be able to comprehend the existing software in order to perform analysis activities [20] such as defect analysis and impact analysis. All six of the CK Metrics discussed earlier have a great impact on analyzability. From lack of cohesion to highly coupled modules to deep inheritance trees, all of these factors contribute to the difficulty of reading, tracing, and comprehending the code base. Some options for measuring analyzability are audit trail capability, diagnostic function support, failure analysis capability, failure analysis efficiency, and status monitoring capability [19].

Changeability

Changeability metrics relate to the actual maintenance activities that modify the existing code. The modification could be the result of discovering a defect, the need to adapt to a new environment, or the request for a new enhancement. Regardless of the reason for the change, most of the same qualities that make a code easier to analyze also makes it easier to change. High cohesion and low coupling are critical when developing an application with a focus on maintenance. An organization should review the relationship between internal metrics and changeability by gathering some of the following external metrics: change cycle efficiency, change implementation elapse time, modification complexity, parameterized modifiability, and software change control capability [19].

Stability

Most users dislike unexpected software behavior. Stability related metrics such as change success ratio and modification impact localization [19] provide evidence of unexpected behavior or the lack thereof. One method for decreasing the risk of unexpected behavior is to complete a thorough impact analysis and ensure modified code paths are systematically tested. With this understanding it is clear that the internal metrics that impact analyzability and testability are also crucial to predicting software stability.

Testability

The final maintainability component discussed is testability. As its name suggests, testability refers to the ease at which the software can be tested, or verified and validated. To measure testability, an organization should collect the following measurements: availability of built-in test function, re-test efficiency, and test restartability [19]. Coupling is a critical characteristic when determining testability [21]. When modules are coupled, it is more difficult or impossible to isolate the module under test to ensure the test is focusing only on the desired code and producing clear concise results. CBO and RFC metrics will have a strong correlation to the external metrics related to testability.

Conclusion

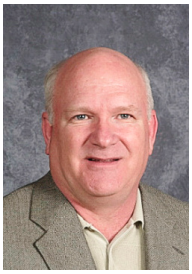
Software operations and maintenance is a critical phase in the software lifecycle. While much emphasis is placed on the development phase, most software products do eventually reach the operations and maintenance phase where they must then be maintained until the system is no longer needed or is replaced by a new product. It stands to reason, then, that software should be developed with maintenance in mind. But this is not typically the case because the software development team is focused on quickly creating a product to meet the needs of the customer at a cost and schedule the customer is willing to accept. The team is likely not considering how the product will be maintained after it is in operation.

However, when the same organization supports both software development and software operations and maintenance, there is incentive to keep maintenance in mind during the development process. It is also easier to capture both internal metrics, such as CK Metrics, that are available during the development phases as well as external metrics which are available only after the product is in operation. Internal metrics can help predict the level of support required to maintain the product once it is in operation. Furthermore, by analyzing the relationships between these two sets of metrics, the organization can improve both development and maintenance processes and thus improves the overall quality of the product as it supports the software from the cradle to the grave.

ABOUT THE AUTHORS

Jennifer Walters is a software development analyst at Northrop Grumman Enterprise Shared Services. She holds a B.S. in Computer and Information Science from University of Maryland University College, an M.A.E.D. in Secondary Education from University of Phoenix, and an M.S. in Information Technology from University of Maryland University College.

RR 2 BX 542
Nelson Drive
Ridgeley, WV 26753
E-mail: jennifer.walters@ngc.com
Phone: 304-726-4174 (home)



Dr. Kevin MacG. Adams is an Adjunct Professor at the University of Maryland University College where he teaches software and systems engineering in the graduate program in Information Technology. Dr. Adams is a retired Navy submarine officer and information systems consultant. Dr. Adams holds a B.S. in Ceramic Engineering from Rutgers University, an M.S. in Naval Architecture and Marine Engineering and an M.S. in Materials Engineering both from MIT, and a Ph.D. in Systems Engineering from Old Dominion University.

University of Maryland University College
3501 University Blvd. East,
Adelphi, Maryland 20783
E-mail: kevin.adams@faculty.umuc.edu
Phone: 757-855-1954 (home)

REFERENCES

1. IEEE, IEEE and ISO/IEC Standard 24765: Systems and software engineering – Vocabulary. New York and Geneva: Institute of Electrical and Electronics Engineers and the International Organization for Standardization and the International Electrotechnical Commission, 2010.
2. A. Abran and J. Moore, Guide to the Software Engineering Body of Knowledge, 2004 ed. Los Alamitos, CA: The Institute of Electrical and Electronics Engineers, 2004.
3. J. Bansiya and C. Davis, "A hierarchical model for object-oriented design quality assessment," IEEE Transactions on Software Engineering, vol. 28, pp. 4-17, 2002.
4. M. Barbacci, M. Klein, T. Longstaff, and C. Weinstock, "Quality Attributes (Technical Report: CMU/SEI-95-TR-021, ESC-TR-95-021)," Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA1995.
5. E. Bagheri and D. Gasevic, "Assessing the maintainability of software product line feature models using structural metrics," Software Quality Journal, vol. 19, pp. 579-612, 2011.
6. M. Bocco, D. L. Moody, and M. Piattini, "Assessing the capability of internal metrics as early indicators of maintenance effort through experimentation," Journal of Software Maintenance & Evolution: Research & Practice, vol. 17, pp. 225-246, 2005.
7. N. Nagappan, "Toward a software testing and reliability early warning metric suite," in Proceedings of the 26th International Conference on Software Engineering, ed Los Alamitos, CA: IEEE Computer Society, 2004, pp. 60-62.
8. M. R. J. Qureshi and W. Qureshi, "Evaluation of the design metric to reduce the number of defects in software development," International Journal of Information Technology and Computer Science, vol. 4, pp. 9-17, 2012.
9. S. R. Chidamber and C. F. Kemerer, "Towards a metrics suite for object oriented design," in Proceedings of the Conference on Object-oriented programming systems, languages, and applications (OOPSLA 91), A. Paepcke Ed., ed New York: Association for Computing Machinery, 1991, pp. 197-211.
10. R. C. Martin, Agile Software Development: Principles, Patterns, and Practices. Upper Saddle River, NJ: Prentice-Hall, 2003.
11. T. J. McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, vol. SE2, pp. 308-320, 1976.
12. T. J. McCabe and C. W. Butler, "Design Complexity Measurement and Testing," Communications of the ACM, vol. 32, pp. 1415-1425, 1989.
13. P. M. Shanthi and K. K. Duraiswamy, "An empirical validation of software quality metric suites on open source software for fault-proneness prediction in object oriented systems," European Journal of Scientific Research, vol. 51, pp. 166-181, 2011.
14. V. Basili, L. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," IEEE Transactions on Software Engineering, vol. 22, pp. 267-271, 1996.
15. K. E. Emam, W. Melo, and J. Machado, "The prediction of faulty classes using object-oriented design metrics," Journal of Systems and Software, vol. 56, pp. 63-75, 2011.
16. W. Li and S. Henry, "Object oriented metrics that predict maintainability," Journal of Systems and Software, vol. 23, pp. 111-122, 1993.
17. J. M. Veiga and M. J. Frade, "Treecycle: a Sonar plugin for design quality assessment of Java programs (Technical Report: CROSS-10.07-1)," Fundação para a Ciência e a Tecnologia, Centro de Ciências e Tecnologias de Computação, Braga, Portugal2010.
18. G. Vandana, K. K. Aggarwal, and Y. Singh, "A fuzzy approach for integrated measure of object-oriented software testability," Journal of Computer Science, vol. 1, pp. 276-282, 2005.
19. ISO/IEC, Software engineering - Product quality - Part 2: External metrics (ISO/IEC TR 9126-2). Geneva: International Organization for Standardization and the International Electrotechnical Commission,, 2003.
20. T. M. Pigoski, Practical Software Maintenance: Best Practices for Managing Your Software Investment. New York: John Wiley & Sons, Inc. , 1997.
21. S. Khatri, R. S. Chhillar, and A. Sangwan, "Analysis of factors affecting testing in object oriented systems," International Journal on Computer Science & Engineering, vol. 3, pp. 1191-1196, 2011.