

A Change Impact Dependency Measure for Predicting the Maintainability of Source Code

Franck Xia, Praveen Srikanth

Computer Science Dept., University of Missouri-Rolla, Rolla, MO 65409; xiaf@umr.edu

1. Introduction

Software maintenance costs far more than development in software industry. Planning, managing, and controlling maintenance is an important issue that software engineers must deal with everyday. However, as maintenance heavily relies on the knowledge and skills of maintainers, maintainability is related to many subjective and vague factors which are not easy to treat rigorously. It is thus difficult for us to design and model a measure such that a calculation based on some internal software artifacts could reflect the external view of software maintainability. In this paper, we first articulate the theoretic difficulties with the existing metrics designed for predicting software maintainability. To overcome the difficulties, we propose to measure a purely internal and objective attribute of code, namely change impact dependency, and show how it can be modeled to predict real change impact. The proposed base measure can be further elaborated for evaluating software maintainability.

2. Challenge with Quality Measurement

Dependency is an essential aspect to consider for designing the architecture of complex systems and it directly affects the maintainability and many other quality attribute of software. But the notion of dependency itself is vague and subject to different interpretations, i.e. static, dynamic, one-way change impact or two-way mutual dependency. This ambiguity creates confusion when we strive to measure maintainability. Coupling is an internal attribute commonly used for evaluating dependency and maintainability. However, when considering the maintainability of a component, only one-way change impact dependency is relevant, two-way dependency is not. Some well-known coupling metrics, e.g. coupling between objects [1] do not make such a crucial distinction. One consequence of this conceptual ambiguity is that existing dependency metrics cannot stand a basic theoretic validation. By measurement theory, a measure must satisfy the representation condition which means, in plain terms, that a heavy man must have a large weight number on a correctly built scale [2]. Theoretically, we must reject a measure if it cannot preserve the empirical relations we know about the ranking of the entities to be measured. One elementary property of change impact dependency is that it could be transitive: if A depends on B and B on

C, then A may depend on C [3]. But all the known coupling metrics count only the direct change impact and ignore the indirect ones or ripple effect [1,4], which violates the transitivity property and its entailed ranking. Tracing ripple effect is essential for maintenance [3], but none of the existing code maintainability metrics consider ripple effect either [5-7]. The fact that we do not have a sound measure for some crucial quality concepts, i.e. coupling and dependency, suggests that more research effort is needed for developing sound software measures.

3. Change Impact Dependency (CID)

Our goal is to develop a measure based on which we can estimate the change impact (CI) when one system component is to be modified. This measure will eventually enable us to predict the cost of maintenance. We assume that we are limited to study source code without any knowledge about the types of change, i.e. deletion, addition or modification, nor on which part of code, i.e. data or program logic, the change will occur.

3.1 Identifying Fundamental Factor

Based on our general assumption, we focus on tracing the static CID. When a change occurs in a statement s , what we want to know is what are the other statements that could be potentially affected. Although the change in s may affect either data or program control part, for the impact of change, we only need to trace the change of data, whether the change is about name, type, address, value, scope, lifetime, or structure of data/class. This is because when a change affects a program control construct in s which contains no assignment operator, there will be no further impact; otherwise, the left hand side (LHS) variable x of s will be affected, and to trace the impact of change in s we must trace the impact of change on x . Tracing CID through sub-program or method call is to tracing the impact of change via actual parameters. We can demonstrate that all kinds of coupling described in OO literature [4] can be reified though the most fundamental data change.

3.2 Algorithm for Tracing Potential Change Impact

Given any system component M , changing any part of M generally first affects the subsequent part S of M and then through one sub-program call to another in a decomposed system, propagates far beyond the immediate super-ordinate and sub-ordinates of M . In S

$\subset M$, given a change data d , our algorithm first traces all assignment statements and conditions in S containing d . The tracing of the potential CI depending on d stops if d appears at the LHS of an assignment, otherwise, the LHS variable of the assignment becomes a new change data that needs to be traced for the indirect impact from d . When d is an actual parameter in a sub-program call, we continue to trace the ripple-effect of change beyond M . This tracing algorithm is carried out recursively till no other sub-program is called. The impact to the superordinates of M is traced through the output parameters of M . We select LOC (line of code) as a measurement unit for CI. For any sub-program M , by the end of tracing, the potential CID ($C_{potential}$) from M can be represented by $\{b_0, b_1, b_2, \dots\}$, b_0 representing the number of lines that could *potentially* be affected in M (level 0), b_1 the number of lines traced in sub-programs immediately called in M or in M 's callers (level 1), etc.

3.3 Modeling/Predicting Real Change Impact

Our algorithm traces the maximum potential impact of change from a source sub-program M . In reality, for certain types of change, its real impact may end before our tracing algorithm terminates, the same way as any ripple effect diminishes progressively in a medium when it propagates away from the center of the impact. Thus we propose to model and predicate real CI based on the traced potential CID as follows:

$$C_{predicted} = \alpha \cdot \left[\sum_{i=0}^{\infty} \beta^i b_i \right] \quad (1)$$

In formula (1), α is a coefficient introduced to take into account the fact that not all the lines traced could be, in reality, affected. β is the attenuation coefficient which reifies the attenuation of ripple effect when the impact pass through one sub-program to another. As α and β are unknown, test cases are designed and real changes are made. Based on a set of real change data ($C_{Actual(k)}$ standing for the total number of lines modified in the k^{th} test case), we have estimated α and β through minimizing the mean square errors of $\sum_k \Delta_k^2$, with

$$\Delta_k = \alpha \cdot \left[b_0 + \sum_{i=1}^{\infty} \beta^i \cdot b_i \right] - C_{Actual(k)} \quad (2)$$

Note that our model is non-linear with respect to α and β . No existing regression models can be directly used in our case and the estimation task is non-trivial. By limiting to two levels of ripple effect, an analytic solution has been found. The formulas for α and β are quite complex and not represented in this short abstract. With our test cases, we obtain $\alpha \approx 1.00$ and $\beta \approx 0.90$. Note that the two programs used for our model validation have been taken from websites providing

open source code. The maintenance activities are designed and performed on these two source programs.

3.4 Modeling Change Impact in OO Source Code

Our CID model is general and hence can be applied to either structured or object-oriented code, for a method is just a specific sub-program with a scope limitation. With the estimated α and β , formula (1) can be used for predicting real change impact based on traced potential change impact. However, when dealing with OO software, the unit of interest is class rather than method. So far our CID model does not make such a distinction. We now propose a more refined model for OO code, decomposing potential CID into intra-class and inter-CIDs:

$$C_{potential} = C_{intra-class-CID} \cup C_{inter-class-CID}$$

For predicting real CI, formula (1) is applied only to the inter-class part of the potential CID ($C_{inter-class-CID}$). Note that data to be change in OO code could be non-local for a method, such as an object attribute or data in a super-class. These specificities can be treated as shared data with our general model without difficulty.

4. Conclusion

Our change impact dependency model is theoretically sound, for it respects our empirical observation about CID. The proposed measure for predicting CI helps us to estimate the real impact of change during the maintenance phase. By integrating other factors that have been already considered in the existing methods, such as comment lines in each component [5] or the probability of change for each component, we are confident that a more reliable measure for predicting software maintainability can be obtained. Finally, through reliable cost/effort estimation, we can estimate, plan, and manage maintenance before the start of the maintenance process, thereby the maintenance activity can be efficiently executed in an organized manner.

Reference

1. Chidamber S.R. Kemerer C.F., Towards a metrics suite for object-oriented design, IEEE T-SE, 20(6), 1994, 476-493
2. Krantz D.H., Luce R.D., Suppes P., and Tversky A., Foundations of Measurement, vol.1, Academic Press, 1971
3. Arnold R.S. & Bohner S.A., Software Change Impact Analysis, Wiley, 1996
4. Briand L. et al., A unified framework for coupling measurement in object-oriented systems, IEEE T-SE, 25, 1, 1999, 91-121
5. Coleman, D. et al., Using Metrics to Evaluate Software System Maintainability, IEEE Computer, 27(8), 1994, 44-49
6. Munson J., Elbaum S.G., Code Churn: A Measure for Estimating the Impact of Code Change, Proc. ICSM, 1998
7. Polo M. et al., Using Code Metrics to Predict Maintenance of Legacy Programs: A Case Study, Proc. ICSM, 2001