Object oriented metrics useful in the prediction of class testing complexity

Ilona Bluemke
Institute of Computer Science
Warsaw University of Technology
Nowowiejska 15/17, 00-665 Warszawa, Poland
I.Bluemke@ii.pw.edu.pl

Abstract

Adequate metrics of object-oriented software enable one to determine the complexity of a system and estimate the effort needed for testing already in the early stage of system development. The metrics values enable to locate parts of the design that could be error prone. Changes in these parts could significantly improve the quality of the final product and decrease testing complexity. Unfortunately, only few of the existing Computer Aided Software Engineering tools (CASE) calculate object metrics. In this paper methods allowing proper calculation of class metrics for some commercial CASE tool have been developed. New metric, calculable on the basis of information kept in CASE repository and useful in the estimation of testing effort have also been proposed. The evaluation of all discussed metrics does not depend on object design method and on the implementation language.

1. Introduction

The use of object oriented software development techniques add new elements to software complexity in the software development process and in the final product. The traditional metrics for measuring software like LOC (lines of code) are inadequate for case of object oriented software and can be applied only to the measurements of the complexity of methods in a class, like McCabe cyclomatic complexity. In recent years many researchers and practitioners have proposed metric for object oriented software [1,3,6,7,8,10].

Most of the research efforts in evaluating the complexity of a software system have been applied at the final stages of the software production. Almost all research has been dedicated associating with source code a complexity number based on some rules. Often these rules depend on programming language. Some metrics proposed for C++ programs can be found in [1, 9]. The determination of the complexity for an already written code may be too costly because the latter the changes are introduced the more expensive they are. The complexity number can be associated either with a class or with the whole project. Programmers can use class-level metrics to identify error prone classes. The class-level metrics can be used to estimate testing effort, the possibility of code reuse, and to improve the quality of the class code as well. In section 2 some “class-level” metrics, which can be used to predict testing complexity, are described. The brief survey of “system-level” metrics can be found in [4].

Availability of a suitable and adequate measuring tool already at the early stage of a program development enables early prediction of the system complexity thus reducing the cost of making necessary changes.

Since the 1980s a wide range of tools to support software development have been constructed. CASE technology can be applied in a cost-effective way to significantly reduce software costs and development time and to increase the quality of software. CASE workbenches are now available as a support for most software process activities. Workbenches supporting analysis and design are widely used in industry. The workbenches may support specific design and analysis method or may constitute more general diagram editing systems with the knowledge of the most common methods [16]. A good example of such workbenches is Paradigm Plus, a registered trademark of Platinum Technology, supporting many object-oriented analysis and design methods (UML, OMT, Booch, Coad-Yourdon, Shlaer-Mellor).

In a CASE tool used for analysis and design there are normally components such as: structured diagramming, data dictionary, report generation, design checking, import/export facilities, code generators, repository query facilities. These components are integrated through a shared repository whose structure is proprietary to the vendor and therefore its environment is usually closed. It is difficult for the users to modify the tools or to add their own tool. In Paradigm CASE a language [15] has been provided which enables access to almost all information kept in the repository.
Most of the metrics proposed by Chidamber-Kemerer [6] and some metrics proposed by Bashir-Goel [1] have been implemented in Paradigm script language. A new metric, suitable for the prediction of testing effort, has been proposed in section 4 and its code written in Paradigm script language is also given in Appendix 1. The above mentioned metrics can be used to identify error prone classes, to determine testing effort and to allow evaluating the possibility of code reuse. They can be used to reduce the complexity of the design at early stages and do not depend on the implementation language used.

Paradigm enables the generation of code in many languages (C++, Java, SQL, etc) but similarly to other CASE tools (eg. Objectiff, Select, Rose) does not enable calculation of object-oriented metrics. To our best knowledge only one CASE tool, OODesigner [14], capable of evaluating useful metrics has been developed in Korea. The scripts, evaluating metrics, can be easily used in the software development supported by Paradigm or other object CASE tools providing a language to extract information from the repository. They can significantly help the development team in finding software fragments difficult to test and error-prone classes.

2. Class metrics

Almost all research in the field of object-oriented metrics has been directed towards association with source code a complexity number. However class complexity evaluation after a class code had been already written may be too expensive. In a widely cited paper Chidamber and Kemerer [6] introduced six object-oriented metrics based upon measurement theory. The proposed set of metrics is as follows:

1) number of children - NOC,
2) depth of inheritance tree - DIT,
3) weighted methods per class - WMC,
4) response for a class - RFC,
5) coupling between objects - CBO,
6) lack of cohesion in methods - LCOM.

In their paper Chidamber-Kemerer [6] also provide an analytical confrontation of their proposed metrics with Weyuker’s [17] list of measurement principles.

The NOC metric is defined as the count of immediate subclasses. With greater NOC the likelihood of improper abstraction of parent class is greater. The NOC may also give an idea of the potential influence a class has on the design. If a class has a large NOC value, it may require more testing of the methods in that class.

DIT is the length of the maximal path to the root of the class hierarchy. It can be observed that the deeper a class in the hierarchy, the greater can be the number of inherited methods making it more difficult to predict its behaviour so more effort is needed in testing this class. Deeper trees constitute greater design complexity, since more methods and classes are involved.

WMC is the count of methods in a class. The number of methods and its complexity is a predictor of how much time and effort is required to develop and maintain the class. Classes with large number of methods are likely to be more application specific, limiting the possibility of reuse. The more methods defined in a class the greater is its possible impact on children, since children inherit all defined methods.

The RFC metric is the cardinality of the set of all methods that can be potentially executed in the response to the arrival of a message to an object. RFC is a measure of the potential communication between the class and another classes. If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding required on the part of the tester. The larger the number of methods that can be invoked from a class, the greater the complexity of the class. A worst case value for possible responses will assist in appropriate allocation of testing time. RFC does not count calls to X- library functions and I/O functions like printf, scanf that are present in C++, interface classes are counted in RFC.

The CBO metric is approximately defined as the number of couples with other classes (where calling a method or using instance variable from another class constitutes coupling). The more independent a class is, the easier it is to reuse it in another application. To improve modularity and promote encapsulation inter class coupling should be small. The larger the number of couples, the higher is the sensitivity to changes in other parts of the design and therefore maintenance is more difficult. CBO metric is useful to determine how complex the testing of various parts of a design is likely to be. The higher the CBO is, the more rigorous the testing needs to be. CBO depends on the manner in which methods are designed and not on the functionality of the class.

The LCOM metric is defined as a count of the method pairs that do not have common instance variable minus the count of method pairs that do. The larger the number of similar methods, the more cohesive is the class. If none of the methods of a class display any instance behaviour, i.e. do not use any instance variables, they have no similarity and the LCOM for the class will be zero. Cohesiveness of methods within a class is desirable, since it promotes encapsulation. Lack of cohesion implies classes should be probably split into two or more subclasses. Any measure of disparity of methods helps identify flaws in the design of classes. Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.
Bashir and Goel also proposed [1] a set of metrics for measuring the complexity of C++ classes and some of them can be applied at an early stage of software development. Some of Bashir/Goel metrics are similar to Chidamber-Kemerer metrics. The proposed set of metrics is composed of context coupling per class, import/export ratios, visibility, internal complexity, volatility and reuse. Context coupling describes the interconnection relation between different classes, which is an indication of the design complexity. Context coupling for a class is the number of other classes that use the offered service. Small, one to two context coupling per class, indicates good class design and small dependency between classes. Such design should lead to smaller number of defects, and the complexity of the testing process will be low.

There are obviously more papers on object oriented metrics in the literature. Due to the space limitations they are not described here, but the reader can find abstracts of more than 500 papers on this subject in http://dec.bournemouth.ac.uk/ESERG/bibliography.html.

3. Metrics calculation on class diagram

The following three Chidamber-Kemerer metrics: NOC, DIT and WMC can be easily calculated from the class diagram. In Fig. 1 a simple class diagram in UML notation [5] is shown. From this diagram the maximal depth of inheritance tree DIT=2, and the numbers of children for each class (NOC=1 for class A, NOC=2 for class B, for other classes NOC=0) can be easily found.

Although the values of DIT and NOC can be calculated at initial stages of the design (only class diagram with identified classes and relationships is necessary) they provide valuable information about design complexity and testing efforts.

The WMC metric (the count of methods in a class) can be calculated as soon as class operations are identified.

For class diagram shown in Fig.1, WMC=3 for class B and for other classes WMC=2.

![Class Diagram](image)

The RFC metric can also be calculated at the same stage of project development. The RFC metric for class B is 5 since the arrival of messages to this class could possibly cause the execution of 5 methods (3 declared in class B and 2 inherited from class A). For class E the RFC=4 because two methods are declared in class E and
two methods are potentially visible through an association link from class A and F (public methods oper1A and oper1F). The method oper2A in class A, is protected so it cannot be executed by class E, which is not a subclass of A.

To calculate remaining Chidamber-Kemerer metrics (CBO – coupling between objects, lack of cohesion in methods - LCOM) the source code of methods is needed. Since at early stages of the design process the source code of methods is not available it was proposed in [3] to substitute the CBO metric by a new metric ABC – association between classes.

ABC – association between classes is a measure of a potential coupling through association links. In other words it is a number of methods and class attributes from associated classes which can be reached. To calculate ABC metric class diagram with attributes and operations is necessary but the code of methods need not be available. In Fig. 1 a class diagram with association relationships is presented. From this diagram ABC measure for all classes can be evaluated. For class E ABC=4 (association with class F and through this link two attributes and an operation from class F are potentially visible, association with class A and one public operation through this link can be reached). For class F ABC=4 (one association with class E and through this link one attribute and one operation from class E are potentially visible, through association with class G one attribute and one operation from class G are potentially coupled). The ABC metric, with some similarities to Bashir-Goel context coupling metric and Chidamber-Kemerer CBO metrics, will be always bigger than context coupling because visible attributes are counted as well, and potential, not real, coupling is counted. However ABC metric resembles Chidamber-Kemerer’ CBO metric and Bashir–Goel’s context coupling metric it does not need the code of class operations, as these two do, so it can be calculated earlier and does not depend on the implementation language.

ABC metric can replace CBO metric in the prediction of testing effort. The higher the ABC value is, the more effort is needed in testing. Increased values of ABC show higher sensitivity to the changes in the design and therefore more difficult maintenance.

4. Length of message sequence metrics

In testing object-oriented software, especially while integrating classes, often sequences of messages are executed (eg. methods-message paths by Jorgensen and Erickson [11], methods sequence method by S. Kirami and W. T. Tsai [12]). The sequences of messages passed between objects are shown in UML [5] sequence diagrams. The lengths of message sequences could be the basis for the prediction of testing complexity. The longer the sequences are, the more time is needed for testing the integration of objects.

Length of message sequence (LMS) metric is counting the number of messages in sequences of messages. An example of a sequence diagram is shown in Fig.2.

Different lines are used in Fig.2 to show different message sequences. The lengths of message sequences (the values of LMS metrics) are following:

Sequence 1 = 5
Sequence 2 (denoted on Fig. 2 with dashed line) = 3
Sequence 3 (denoted on Fig. 2 with dotted line) = 4.

Sequences of messages, called Message/Methods Paths (MM-Paths) are used in object-oriented integration testing [11]. MM-Paths are elements of Atomic System Function (ASF). ASF [11] is an elemental function visible at system level. ASF starts with an input event, is followed by MM-Paths and is terminated by output event. ASF are often presented at object interaction diagram [5] (sequence diagram – Fig. 2).

The LMS metrics could be used to predict needed in object-integration testing. The ! shown on sequence diagram, are to be tested and LMS metrics give information about theirs lengths. The bigger

Experiments, showing practical usefulness of LMS metric in the prediction in testing complexity, are under
metrics in the prediction of design and testing complexity. Experiments with ABC and proposed in section 4 LMS metrics) to aid design evaluation is described. Measurements (Chidamber-Kemerer [6] and MOOD [10] design process. In [13] a case study into using demonstrating the utility of metrics in actually aiding the maintainability, etc. There is a lack of published work under development. It is expected that this experiment would give information how the metrics values relate to the efforts really needed in testing.

5. Conclusion

Paradigm, as many other object CASE tools, enables code generation in many languages (C++, Java, SQL, etc) but does not calculate any object-oriented metrics. In this paper an implementation of some object metrics in Paradigm Plus script language was described. The proposal to incorporate metrics at the design stage, within CASE tools, is novel. To the author’s best knowledge only one CASE tool, OODesigner [14] developed in Korea, is capable of evaluating some metrics.

As no conclusion has been reached yet about a good set of metrics for object-oriented systems, a new object-oriented metric LMS –lengths of message sequence, an indication of object-oriented integration testing effort, was proposed and implemented. LMS metric does not need the source code of class operations, so it can be calculated at early stages of software development. The calculation of this metric could be incorporated into some CASE tools as shown in appendix 1.

Implemented scripts evaluating metrics can be easily used in the software development supported by Paradigm or other CASE tools (eg. Select Enterprice) provided with similar language to access information in the repository. They are independent on the design method and implementation language. The metrics can help the development team in finding complex to testing software fragments and error-prone classes.

With the aid of metrics and heuristics the whole architecture – software project or part of it – classes, can be assessed. Then it is redesigned, proposed changes are implemented, and new design is evaluated against the original one. Design heuristics are rules of thumb that can guide designers as they choose between various alternatives, they capture the experience of skilled designers. A high degree of agreement between different authors in the literature might increase confidence in the validity of their heuristics. The utility of metrics will be questioned until sufficient number of empirical validations will be performed. The experiments should enable to establish a relationship between metrics and real quality attributes of a system, such as reliability, testability, maintainability, etc. There is a lack of published work demonstrating the utility of metrics in actually aiding the design process. In [13] a case study into using measurements (Chidamber-Kemerer [6] and MOOD [10] metrics) to aid design evaluation is described. Experiments with ABC and proposed in section 4 LMS metrics in the prediction of design and testing complexity are under development. It is expected that this experiment would give information how the metrics values relate to the efforts really needed in testing.

6. References

7. Appendix 1: Script for LMS metric

Below a script calculating LMS metric described in section 4 is shown in Paradigm script language.

GLOBAL TempMaxY AS Integer
GLOBAL TempObject AS String
DIM MinObject AS String
DIM TempMinY AS Integer
DIM TempNext AS Integer
DIM Result AS Integer

SELECT All FROM Diagram WHERE NAME=Diagram.Name
FOREACH Diagram
  TempMaxY = 0  'searching maximal y axes among messages
  FOREACH Diagram.Item
    IF Item.Relationship = "Object.Message.Object" THEN
      IF TempMax < Item.y THEN
        TempMax = Item.y
      ENDIF
    ENDIF
  NEXT
  TempMinY = TempMaxY
  FOREACH Diagram.Item
    IF Item.Relationship = "Object.Message.Object" THEN
      IF Item.y < TempMinY THEN
        TempMinY = Item.y
        MinObject = Item.Object
      ENDIF
    ENDIF
  NEXT
  TempMinY = NextMessageY (-1)  'first sequence
  TempNext = NextMessage (TempMinY)
  Result = 0
  WHILE TempNext <> TempMinY
    IF IsMessage (TempObject, TempMinY, TempNext) = 1 THEN
      FOREACH Diagram.Item
        IF Item.Relationship = "Object.Message.Object" THEN
          IF (Item.y > TempMinY) AND (Item.y < TempNext) THEN
            Result = Result + 1
          ENDIF
        ENDIF
      NEXT
      PRINT "Sequence – ",Result
      TempMinY = TempNext
    ENDIF
  ENDIF
  TempNext = NextMessageY (TempNext)  'next sequence
NEXT
END

'returns y axes of next message outgoing from the first 'object
FUNCTION NextMessageY (ByVal ObjectY)
DIM TempResult AS Integer
TempResult = TempMaxY
FOREACH Diagram.Item
    IF Item.Relationship = "Object.Message.Object" THEN
        IF Item.Owner = MinObject THEN
            IF (Item.y > ObjectY) AND (Item.y < TempResult) THEN
                TempResult = Item.y
                TempObject = Item.Member
            ENDIF
        ENDIF
    ENDIF
NEXT
NextMessageY = TempResult
END FUNCTION

' function checking if between pairs of messages of an
' object exists outgoing or incoming message to the
' message with the name given as parameter
' returns 1 if message exists, 0 otherwise
FUNCTION IsMessage (ByVal ObjectName As String, ByVal MinY As Integer, ByVal MaxY As Integer) As Integer
    FOREACH Diagram.Item
        IF Item.Relationship = "Object.Message.Object" THEN
            IF Item.Owner = ObjectName THEN
                IF (Item.y > MinY) AND (Item.y < MaxY) THEN
                    IsMessage = 1
                    EXIT FUNCTION
                ENDIF
            ENDIF
        ENDIF
    ENDIF
NEXT
IsMessage = -1
END FUNCTION