

# Experiences of using the Dagstuhl Middle Metamodel for defining software metrics

Jacqueline A. McQuillan  
Department of Computer Science  
National University of Ireland, Maynooth  
Co. Kildare, Ireland  
jmcq@cs.nuim.ie

James F. Power  
Department of Computer Science  
National University of Ireland, Maynooth  
Co. Kildare, Ireland  
jpower@cs.nuim.ie

## ABSTRACT

In this paper we report on our experiences of using the Dagstuhl Middle Metamodel as a basis for defining a set of software metrics. This approach involves expressing the metrics as Object Constraint Language queries over the metamodel. We provide details of a system for specifying Java-based software metrics through a tool that instantiates the metamodel from Java class files and a tool that automatically generates a program to calculate the expressed metrics. We present details of an exploratory data analysis of some cohesion metrics to illustrate the use of our approach.

## Categories and Subject Descriptors

D.2.8 [Software Engineering]: Metrics; D.2.7 [Distribution, Maintenance, and Enhancement]: [Restructuring, reverse engineering, and re-engineering]

## General Terms

Design, Measurement, Standardization

## 1. INTRODUCTION

Software plays a pivotal role in many important aspects of modern daily life. In many cases, if software fails it can have catastrophic consequences such as economic damage or loss of human life. Therefore, it is important to be able to assess the quality of software. Software metrics have been proposed as a means of determining software quality. For example, studies have demonstrated a correlation between software metrics and quality attributes such as fault-proneness [2] and maintenance effort [12].

Many software metrics have been proposed in the literature [5, 13, 9]. In order for these metrics to be widely accepted, empirical studies of the use of these metrics as quality indicators are required. However, there is no standard terminology or formalism for defining software metrics and consequently many of the metrics proposed are incomplete,

ambiguous and open to a variety of different interpretations [4]. For example, Churcher and Shepperd have identified ambiguities in the suite of metrics proposed by Chidamber and Kemerer [5, 6]. This makes it difficult for researchers to replicate experiments and compare existing experimental results and it hampers the empirical validation of these metrics.

Several authors have proposed various approaches for specifying software metrics. Briand et al. propose two extensive frameworks for software measurement, one for measuring coupling and the other for measuring cohesion in object oriented systems [3, 4]. Other approaches include the proposal of formal models on which to base metric definitions and the proposal of existing languages such as XQuery and SQL as metric definition languages [16, 8, 18]. Baroni et al. propose the use of the Object Constraint Language (OCL) and the Unified Modelling Language (UML) metamodel as a mechanism for defining UML-based metrics [1].

In this paper we present details of an approach for specifying Java-based software metrics. This approach is based on one previously proposed by Baroni et al. and involves expressing the metrics as OCL queries over a Java metamodel. We have chosen the *Dagstuhl Middle Metamodel* as our Java metamodel, and we describe the specification of software metrics over this model using OCL. We have implemented a system that supports this approach which provides a flexible and reusable environment for the specification and calculation of software metrics. The system is capable of automatically generating a program to calculate the specified metrics. We have performed a study of several cohesion measures to demonstrate its use.

The remainder of this paper is organised as follows. Section 2 gives details of the approach for specifying software metrics. Details of a system that implements this approach are presented in Section 3. In Section 4, we present an exploratory data analysis of some cohesion metrics. Section 5 gives conclusions and discusses future work.

## 2. DEFINING METRICS

In this section, we give details of an approach for specifying Java based software metrics that is based on the use of metamodels and the OCL.

As the name suggests, a *metamodel* is a model that describes other models. Typically, we think of a model of a software system as being a design model, such as UML class or sequence diagrams, or an implementation model, such as an actual program. A metamodel then would describe

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPPJ 2006, August 30 - September 1, 2006, Mannheim, Germany  
Copyright 2006 ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

```

-- Returns the RFC value for the Class c
context ckmetricset::RFC(c:DMM::Class) : Real
body: self.implementedMethods(c)->union(self.methodsDirectlyInvoked(c)->asSet()->size()

-- Returns a set containing all methods directly invoked by all the implemented methods of Class c
def: methodsDirectlyInvoked(c:DMM::Class) : Set(DMM::Method)
  = self.implementedMethods(c)->collect(m:DMM::Method | self.methodsDirectlyInvoked(m))
    ->flatten()->asSet()

-- Returns a set containing all methods directly invoked by the Method m
def: methodsDirectlyInvoked(m:DMM::Method) : Set(DMM::Method)
  = m.invokes->select(be:DMM::BehaviouralElement | self.isKindOfMethod(be))
    ->collect(belem:DMM::BehaviouralElement | belem.oclasType(DMM::Method))->asSet()

```

**Figure 1: RFC Metric Definition using the DMM.** *This OCL specification defines an operation to calculate the RFC metric for a class, as well as two auxiliary operations. The entities used in the definition are from the DMM.*

the allowable constructs in these models and the allowable relationships between these constructs.

OCL is a standard language that allows constraints and queries over object oriented models to be written in a clear and unambiguous manner [17]. It offers the ability to navigate over instances of object oriented models, allowing for the collection of information about the navigated model. Baroni et al. propose expressing design metrics as OCL queries over the UML 1.3 metamodel [1]. Their approach involves modifying the metamodel by creating the metrics as additional operations in the metamodel and expressing them as OCL conditions.

We have already extended the approach of Baroni et al. to the UML 2.0 metamodel in a manner specifically designed to be reusable for other metamodels [14]. Our extension involves decoupling the metric definitions from the metamodel by creating a metrics package at the meta level. Defining a new metrics set is then a three step process. First a class is created in the metrics package corresponding to the metric set. Then, for each metric, an operation in the class is declared, parameterised by the appropriate elements from the metamodel. Finally the metrics are defined by expressing them as OCL queries using the OCL body expression.

## 2.1 Selecting a metamodel

In order to adapt this approach to specify Java based metrics, it is necessary to have a model of Java programs i.e. a Java metamodel. There is no standardised Java metamodel, but a number of task-specific metamodels have been proposed for various purposes. In order to maximise the reusability of our metrics, we have chosen to use the *Dagstuhl Middle Metamodel* (DMM) as a basis for defining our metrics [11]. The DMM was designed as a schema for reverse engineering that would facilitate interoperability between tools as an agreed exchange format. It is a “middle” metamodel in so far as it seeks to be more abstract than a syntax graph, but less abstract than a high-level architectural description.

The DMM itself is language independent, but contains many features commonly found in languages such as C, C++, Java and Fortran. We do not have space here to reproduce the elements of the model, necessary for a full understanding of our metrics, but details can be found in [11]. We have used the Chidamber and Kemerer (CK) metrics suite [5] to illustrate the approach outlined in this paper. We have successfully expressed the CK metrics as OCL queries over

classes from the `ModelObject` hierarchy of version 0.007 of the DMM. This required approximately 29 OCL queries in total.

As an example of a metric definition, Figure 1 presents the definition of the response set for a class (RFC) metric. The response set for a class is the set of all implemented methods of this class and all methods invoked by this class. The definition is parameterised by a single `Class` from the DMM hierarchy, and the body of the definition returns the size of the response set for this class. The auxiliary operation `methodsDirectlyInvoked(DMM::Class c)` gathers all methods invoked by each of the implemented methods in the class. The operation `methodsDirectlyInvoked(DMM::Method m)` traverses the `invokes` association in the DMM to gather all `BehaviouralElements` invoked by the method `m` and then selects all elements from this set that are `Methods`.

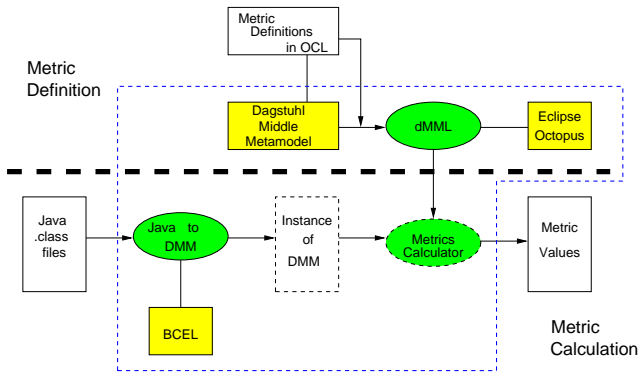
## 3. IMPLEMENTATION

In this section we describe the implementation of our system to calculate metrics for Java programs based on the DMM. This was a three step process:

- Step 1: Create a representation of the classes and associations of the DMM in Java
- Step 2: Develop a tool to convert Java programs to instances of the DMM
- Step 3: Develop a tool that can apply metrics defined in OCL to the instance of the DMM produced in step 2.

Step 1 is easily achieved by depicting the DMM as a UML class diagram, and then using the Octopus [10] tool to generate the corresponding Java classes. We implemented the 19 classes from the DMM `ModelObject` hierarchy directly, and chose to implement the relationships using attributes of these classes, rather than association classes. This implementation decision was made as the explicit relationship classes were not required by our tool. For similar reasons, we did not implement the classes in the `SourceObject` hierarchy that represent details about the code as it appears in the original program. This does not preclude these classes being added later.

The combination of the tools used in our approach is shown in Figure 2. The figure is divided into two layers: the upper layer represents the metric definition process, which



**Figure 2: The use of dMML to define and calculate metrics for Java programs. dMML is part of a tool-chain that calculates metric values from Java .class files.**

is done once for each metric set. The lower layer represents the metric calculation process, where the metrics are applied to a set of Java programs. The main tools we developed are shown as green ovals: dMML for metric definitions, and *Java to DMM* for converting class files to instances of the DMM. The *Metrics Calculator*, also shown as a green oval, is a Java program automatically generated by dMML for each metric set. The third-party software used in our metric definition system is shown by yellow boxes in Figure 2. BCEL is used by the *Java to DMM* tool, and the Octopus plug-in for Eclipse is used in defining the metrics, as described in the following two sub-sections. The definition of the DMM is represented as a UML class diagram, and the corresponding Java representation is forward-engineered using Octopus.

The blue dashed line in Figure 2 delimits the system, and shows that its inputs are a set of metric definitions in OCL and a set of Java programs. The output of the system is the set of metric values calculated by applying the metrics to the Java programs.

### 3.1 Converting Java programs to an instance of the DMM

In order to complete Step 2, it is necessary to read in Java programs and to instantiate the DMM classes produced in Step 1. We chose to process a compiled .class file directly as the contents of the .class file most closely resembled the information needed to instantiate the DMM implementation. In particular, access relationships between classes arising from the use of fields and variables in a method are easy to identify at the bytecode level, since they are translated into a single bytecode instruction.

Our implementation uses the Apache Bytecode Engineering Library (BCEL) to read in and traverse the contents of the .class file. The BCEL API provides classes representing the contents of the .class file, and methods to access classes, fields, methods and bytecode instructions. Using the BCEL it was relatively easy to traverse these structures and instantiate the DMM, and required less than 600 (non-blank, non-comment) lines of Java code. It should be noted that using BCEL would not be suitable for a more detailed representation than the DMM `ModelObject` hierarchy. Source level details such as Java statements (e.g. while and for

loops) are not represented in the bytecode, and tables giving local variable names and mappings to lines of Java code are optional at the .class file level.

### 3.2 Implementing the metric definitions

To complete step 3, we extended a prototype tool dMML (for *Defining Metrics at the Meta Level*) that was first applied to the UML 2.0 metamodel [14]. Our tool is implemented as a plug-in for the integrated development environment Eclipse.

In step 3, a set of metrics are created and defined for the language under consideration which in this case is Java. To achieve this, the language metamodel, DMM is provided along with the metric definitions expressed as OCL queries over this metamodel. dMML uses the Octopus plug-in to perform syntactic and semantic checks on these OCL expressions.

The dMML tool uses the Octopus plug-in to translate the OCL metric definitions into Java code. A Java program, *Metrics Calculator*, is automatically generated by dMML that will calculate the defined metrics for any instance of the Java metamodel (i.e. Java programs). To perform the metric calculations dMML uses the *Java to DMM* tool developed in step 2 to convert the Java programs to an instance of the DMM. The Java code corresponding to the metric definitions is executed and the results from the metric calculations are exported in text format.

The parameterisation of the dMML tool by both the language metamodel and the definition of the metrics set is an important feature of our approach. To extend our system to work with other language metamodels only step 2 of this process needs to be changed. That is, a new tool would need to be developed to create instances of the language metamodel.

## 4. EXPLORATORY DATA ANALYSIS

In this section we present an exploratory data analysis of cohesion measures in order to demonstrate the feasibility of our approach to defining and implementing software metrics.

Using the procedure described in previous sections we have implemented several cohesion measures from [3] and applied them to programs from the DaCapo benchmark suite, version *beta051009* [7]. This benchmark suite is designed for memory management research, and consists of 10 open-source real-world programs.

In order to provide a meaningful comparison, we have chosen the four cohesion metrics that do not involve indirect comparisons, namely *LCOM1*, *LCOM2*, *LCOM5* and *ICH* [3]. We have included constructors, finalisers and accessor methods as ordinary methods, but excluded attributes and methods that are inherited but not defined in a class. Since metric *LCOM5* involves division, we have excluded those classes that cause a divide-by-zero error, namely classes that contain no attributes, or classes that contain exactly one method definition. A total of 4836 classes in the DaCapo benchmark suite meet these conditions.

Table 1 gives a summary of the values of the four metrics over these 4836 classes. The values of *LCOM1* and *LCOM2* are all positive integers, whereas *LCOM5* is normalised to a real number between 0.0 and 2.0. The measure *ICH* has been negated to facilitate comparison since it measures the degree of cohesion, rather than the lack of cohesion measured

	LCOM1	LCOM2	LCOM5	ICH
Min.	0.0	0.0	0.0000	-3887.0
1st Qu.	2.0	0.0	0.5000	-10.0
Median	11.0	6.0	0.8000	-2.0
Mean	185.5	135.5	0.7063	-20.5
3rd Qu.	52.0	34.0	0.9444	0.0
Max.	110902.0	94039.0	2.0000	0.0

**Table 1: Summary of the values for the metrics applied to 4836 of the classes from the DaCapo suite. This table shows the minimum, maximum and mean data values, as well as those at the first, second (median) and third quartiles.**

	LCOM1	LCOM2	LCOM5	ICH
LCOM1	1.0	0.8597	0.5305	-0.7439
LCOM2	0.8597	1.0	0.6453	-0.6463
LCOM5	0.5305	0.6453	1.0	-0.3739
ICH	-0.7439	-0.6463	-0.3739	1.0

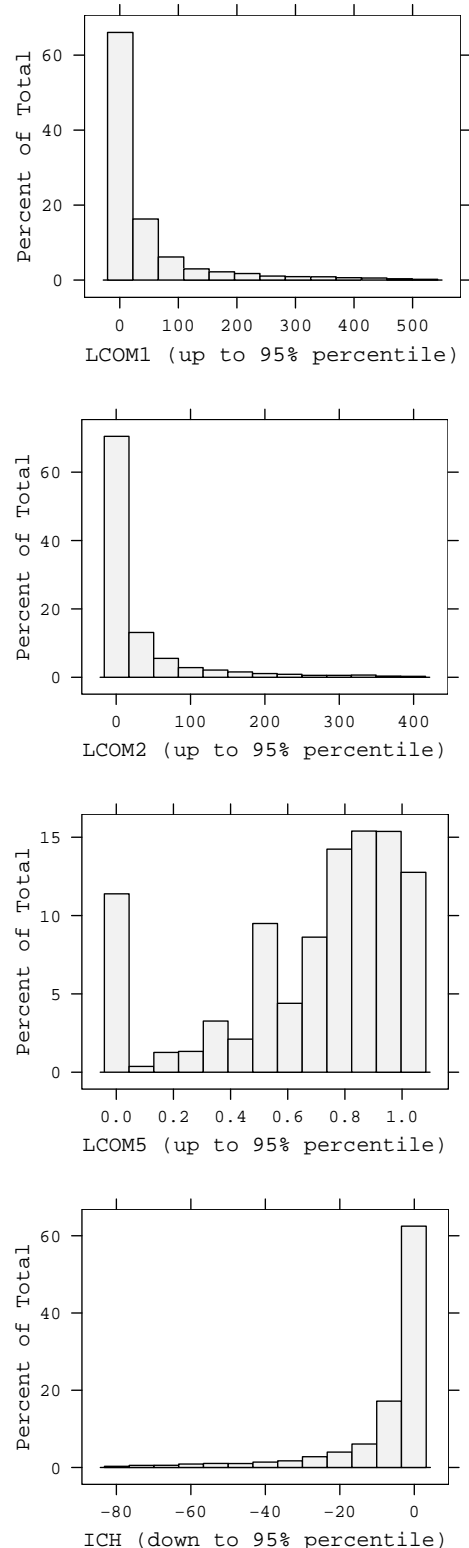
**Table 2: Spearman’s  $\rho$  statistic for each pairing of the four metrics. This statistic compares data sets on a rank basis, with values near 1.0 or -1.0 indicating a strong positive or negative correlation.**

by the *LCOM* metrics. The values of *ICH* are thus all negative integers. That the values shown in Table 1 all fall within these theoretical bounds provides a coarse-grained validation of our implementation.

Figure 3 shows a histogram of the distribution of values for each of the four metrics. In this figure, we have removed the last 5% of data values, including some extreme outliers, in order to better visualise the distribution. The values for *LCOM1*, *LCOM2* and *ICH* are all heavily skewed towards zero, while the normalised values for *LCOM5* reflect a somewhat more even distribution. The distribution of the metric values is consistent with previous work, which showed a similar preponderance of low values for the SPEC JVM98 and JavaGrande benchmark suite [15], and reflects known limitations of these cohesion metrics [2]. The percentage of classes giving a value of zero was 16% for *LCOM1*, 31% for *LCOM2*, 11% for *LCOM5*, and 39% for *ICH*.

In order to check for a relationship between the metric values, the pairwise scatter plots were examined and Spearman’s  $\rho$  statistic was used to estimate the level of association. We have omitted the scatter plots to save space, but the results for Spearman’s  $\rho$  are shown in Table 2. This statistic compares data sets on a rank basis: thus, values close to 1.0 (or -1.0) indicate that the metrics are ranking the classes in the same (or opposite) order. The results in Table 2 confirm the most obvious relationships, i.e. that there is a strong positive correlation between the measures *LCOM1* and *LCOM2*, which have similar definitions, and that each of the *LCOM* measures have a negative correlation with *ICH*, although this is weak for *LCOM2* and *LCOM5*. The strong (negative) association between the values for *LCOM1* and *ICH* is interesting, since these metrics are based on quite different definitions. Further investigation would be required to see if this is a general property of these metrics.

The results presented in this section are exploratory in nature, and only provide a preliminary coarse-grained de-



**Figure 3: Histograms showing the distribution of values for the four cohesion metrics. For clarity, only the first 95% of the values are shown in order to remove extreme outliers.**

scription of the metric values. Nonetheless, we believe that providing such data is important in order to demonstrate the robustness of the metric calculation tool, and as a “smoke test” to ensure that the values are within reasonable boundaries. The design of the dMML tool facilitates the definition of multiple metrics suites, and we hope to exploit this in order to assemble a substantial database of descriptive statistics of object-oriented metrics for benchmark programs.

## 5. CONCLUSIONS AND FUTURE WORK

In this paper we have harnessed the OCL as a language to specify metric definitions over the *Dagstuhl Middle Metamodel*. We have implemented a system that uses the Octopus tool to translate OCL metric definitions into Java code and then calculates the metrics for a Java program. To demonstrate the feasibility of our approach, we have specified four of the alternative definitions of cohesion in OCL and used our system to calculate the metrics for a suite of 10 real-world programs.

In previous work we have applied this approach to defining outline metrics over class diagrams from the standardised UML metamodel [14]. However, metrics defined at the class diagram level cannot evaluate features internal to methods, such as number of method calls etc. A key tenet of our approach is that the range and variance among metric definitions requires a flexible and reusable definition environment.

While developing and implementing the metric definitions and the associated dMML tool, a number of issues arose which we hope to deal with in future work.

- First, despite the formal definition of metrics in [3], there are still some ambiguities for trivial and extreme cases of the metrics, such as when there are no attributes or no methods in a class.
- Second, the correctness of the metric definitions hinges on assumptions made while constructing the metamodel. For example, our tool to translate a class file to an instance of the DMM does not include inherited methods in a class definition. Therefore, it is important that such assumptions would be a known, expressed feature of any metric definition framework.
- Third, the correctness of the program to translate classes to instances of the DMM has not been verified. Errors or omissions at this stage would have a fundamental impact on the correctness of the calculated metrics.

We intend to continue our work by developing a full set of coupling and cohesion metrics, applied to both a Java and the UML metamodel, and to investigate the full potential of modularity and re-usability associated with defining metrics at the meta level.

## 6. REFERENCES

- [1] A. Baroni, S. Braz, and F. Brito e Abreu. Using OCL to formalize object-oriented design metrics definitions. In *ECOOP Workshop on Quantative Approaches in Object-Oriented Software Engineering*, Malaga, Spain, June 2002.
- [2] V. Basili, L. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.
- [3] L. C. Briand, J. W. Daly, and J. K. Wuest. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.
- [4] L. C. Briand, J. W. Daly, and J. K. Wuest. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.
- [5] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.
- [6] N. Churcher and M. Shepperd. Comments on ‘A metrics suite for object-oriented design’. *IEEE Transactions on Software Engineering*, 21(3):263–265, 1995.
- [7] A. Diwan, S. Guyer, C. Hoffmann, A. L. Hosking, K. S. McKinley, J. E. B. Moss, D. Stefanovic, and C. C. Weems. The DaCapo project. <http://www-ali.cs.umass.edu/DaCapo/>, Last accessed July 17, 2006.
- [8] M. El-Wakil, A. El-Bastawisi, M. Riad, and A. Fahmy. A novel approach to formalize object-oriented design metrics. In *Evaluation and Assessment in Software Engineering*, Keele, UK, April 2005.
- [9] N. Fenton and S. Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thompson Computer Press, 1996.
- [10] Klasse Objecten. Octopus: OCL tool for precise UML specifications. Available from <http://www.klasse.nl/octopus/>, Last accessed July 17, 2006.
- [11] T. C. Lethbridge, S. Tichelaar, and E. Ploedereder. The Dagstuhl Middle Metamodel: A schema for reverse engineering. *Electronic Notes in Theoretical Computer Science*, 94:7–18, May 10 2004.
- [12] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111–122, 1993.
- [13] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Prentice Hall Object-Oriented Series, 1994.
- [14] J. A. McQuillan and J. F. Power. Towards re-usable metric definitions at the meta-level. In *PhD Workshop of the 20th European Conference on Object-Oriented Programming*, Nantes, France, July 2006.
- [15] Á. Mitchell and J. F. Power. Run-time cohesion metrics for the analysis of Java programs - preliminary results from the SPEC and Grande suites. Technical Report NUIM-CS-TR2003-08, Dept. of Computer Science, NUI Maynooth, April 2003.
- [16] R. Reißing. Towards a model for object-oriented design measurement. In *ECOOP Workshop on Quantative Approaches in Object-Oriented Software Engineering*, Budapest, Hungary, June 2001.
- [17] J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison-Wesley, 2003.
- [18] F. Wilkie and T. Harmer. Tool support for measuring complexity in heterogeneous object-oriented software. In *IEEE International Conference on Software Maintenance*, Montréal, Canada, October 2002.