# A Definition of the Chidamber and Kemerer Metrics suite for UML [⋆]

Jacqueline A. McQuillan [⋆⋆] and James F. Power

Department of Computer Science, National University of Ireland, Maynooth,
Co. Kildare, Ireland
{jmcq, jpower}@cs.nuim.ie

**Abstract.** Since there is no standard formalism for defining software metrics, many of the measures that exist have some ambiguity in their definitions which hinders their comparison and implementation. We address this problem by presenting an approach for defining software metrics. This approach is based on expressing the measures as Object Constraint Language queries over a language metamodel. To illustrate the approach, we specify how the Chidamber and Kemerer metrics suite can be measured from Unified Modelling Language class diagrams by presenting formal definitions for these metrics using the Unified Modelling Language 2.0 metamodel.

**Keywords:** OO metrics, class diagram metrics, metamodels, UML, OCL.

## 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 [1] and maintenance effort [2].

Many software metrics have been proposed in the literature [3–5]. 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 [6]. For example, Churcher and Shepperd [7] have identified ambiguities in the suite of metrics proposed by Chidamber and Kemerer (CK) [3]. This makes it difficult for researchers to replicate experiments and compare

[⋆] This report is intended to serve as a supplement to the paper ‘Towards re-usable metric definitions at the meta-level’ that appeared in the Postgraduate Workshop of the European Conference on Object-Oriented Programming, Nantes, France, 2006

[⋆⋆] To whom correspondence should be addressed existing experimental results and it hampers the empirical validation of these metrics.

Several authors have attempted to address the problem of imprecise metric definitions. Briand et al. propose two extensive frameworks for software measurement, one for measuring coupling and the other for measuring cohesion in object-oriented systems [6, 8]. Other approaches include the proposal of formal models on which to base metric definitions [9] and the proposal of existing languages such as XQuery and SQL as metric definition languages [10, 11]. 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 [12, 13].

In this report, we take the approach of Baroni et al. [13] and extend it to decouple the metric definitions from the metamodel and thus make the approach generalisable to any metamodel and any set of metrics. Also, in this report we are the first authors to provide a metamodel level definition of the CK metric suite using the OCL and the UML 2.0 metamodel.

The remainder of this report is organised as follows. In section 2, a review of relevant research is presented. In section 3, we give details of an approach that allows for the precise definition of software metrics. In section 4, we illustrate the application of the approach using the CK metrics suite and the UML 2.0 metamodel. Section 5 gives a summary and discussion of future work.

## 2 Related Work

There are many software metrics tools available, most of them based on either conventional metrics such as lines of code, volume or cyclomatic complexity [5], or the Chidamber and Kemerer metrics suite [3]. However, in this section we limit our discussion to those tools and frameworks that explicitly provide for the unambiguous definition of software metrics.

Briand et al. propose an integrated measurement framework for the definition, evaluation and comparison of object-oriented coupling and cohesion metrics [6, 8]. While this framework allows for the unambiguous definition of coupling and cohesion metrics, new frameworks must be developed for other types of metrics.

Harmer and Wilkie have developed an extensible metrics analyser tool for object-oriented programming languages [11]. The tool is based on a general object-oriented programming language metamodel in the form of a relational database schema. Metric definitions are expressed as SQL queries over this schema. The tool is extensible as it has support for incorporating new metrics and new object-oriented programming languages. However, defining the metrics requires the additional effort of the development of C code as well as supplying the SQL queries. In addition, the tool is tied to the underlying metamodel and does not allow the interchange of metamodels.

Another approach put forward by Reißing involved the proposal of a formal model on which to base metric definitions [9]. This model is called ODEM

(Object-oriented DEsign Model) and consists of an abstraction layer built upon the UML metamodel. However, this model can only be used for the definition of design metrics and does not solve the ambiguity problem as the abstraction layer consists of natural language expressions.

El-Wakil et al. propose the use of XQuery as a metric definition language [10]. They propose extracting metric data from XMI design documents, specifically UML designs. XQuery is a language that can be used to extract information from XMI documents. Again this approach has only been used to define metrics at the design level, specifically for UML designs. There is no information available on how it extends to other languages.

Baroni et al. propose the use of the OCL and the UML metamodel as a mechanism for defining UML-based metrics [12]. They have built a library called FLAME (Formal Library for Aiding Metrics Extraction) [14] which is a library of metric definitions formulated as OCL expressions over the UML 1.3 metamodel [15]. Goulão et al [16] have utilised this approach for defining component based metrics and used the UML 2.0 metamodel [17] as a basis for their definitions. We believe that this approach provides a useful mechanism for the precise definition of software metrics and we build upon it in this report.

## 3 Software Metrics at the Meta Level

As we are examining the use of metamodels and the OCL as a basis for the definition of software metrics, we will begin by presenting a short explanation of these concepts.

### 3.1 Metamodels

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 a UML class or sequence diagrams, or an implementation model, such as an actual program. A metamodel then would describe the allowable constructs

| Description | Examples |
|---|---|
| **M3** Meta-metamodel Layer | MOF [18] entities, e.g. Class, Property, Operation |
| **M2** Metamodel Layer | Describes the entities in UML models, e.g. Class, Property, Association |
| **M1** Model Layer | A UML model: actual classes with attributes and associations between these classes. |
| **M0** Data Layer | The instances of a UML model, e.g. objects, method calls. |

**Table 1.** The Four Layer Metamodel Architecture. *This table shows the standard four-layer hierarchy, using UML as an example.*

in these models, for example, the entities that may be depicted in a UML class diagram.

The relationship between models and metamodels is generally depicted as a four-layer hierarchy [19]. The four layers are depicted in Figure 1 in the context of UML. The most abstract layer, $M3$, is the layer that describes the formalism used in modelling languages, the Meta Object Facility (MOF), which is specified as an OMG standard [18]. Beneath this is the $M2$ layer which consists of the metamodel for the language under consideration; common examples include metamodels for UML or Java. The $M1$ layer represents models describing software systems. In Figure 1 the model at layer $M1$ is a user model specified in UML. The $M0$ layer represents the entities that are run-time instances of model elements. The metrics defined in this report are defined at the $M2$ layer, and then applied automatically to UML models at the $M1$ layer.

## 3.2 The Object Constraint Language

The Object Constraint Language (OCL) is a standard language that allows constraints and queries over object-oriented models to be written in a clear and unambiguous manner [20]. It offers the ability to navigate over instances of object-oriented models, allowing for the collection of information about the navigated model.

## 3.3 Extensions to the approach of Baroni et al.

Baroni et al. propose expressing design metrics as OCL queries over the UML 1.3 metamodel [15]. This approach involves modifying the metamodel by creating the metrics as additional operations in the metamodel and expressing them as OCL conditions [12].

We extend this approach by decoupling the metric definitions from the metamodel. This is achieved by creating a separate metrics package at the meta level. Defining a new metrics set is a three step process:

1. A class is created in the metrics package corresponding to the metric set; any auxiliary operations can be defined in this class.

2. For each metric, an operation in the class is declared, parameterised by the appropriate elements from the metamodel.

3. The metrics are defined by expressing them as OCL queries using the OCL `body` expression.

This approach has allowed us to develop an easily extensible tool called dMML (Defining Metrics at the Meta Level) that can be used for specifying software metrics over language metamodels and to automatically generate a program to calculate these expressed metrics. In theory, dMML can be applied to any language metamodel.

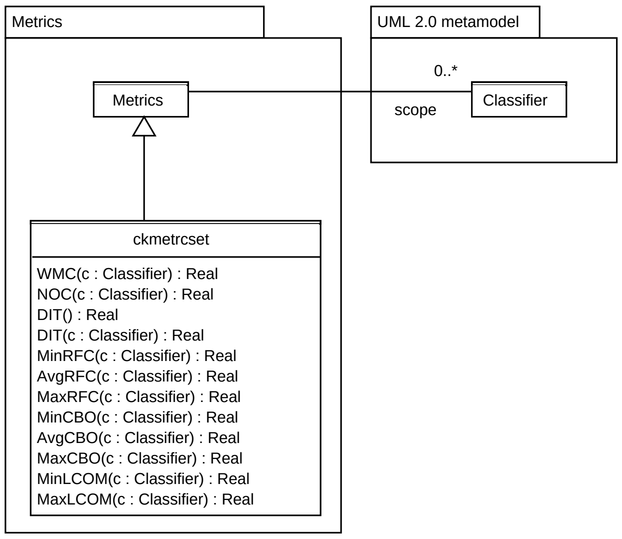
Metrics
UML 2.0 metamodel
Metrics
0..*
scope
Classifier
ckmetrcset
WMC(c : Classifier) : Real
NOC(c : Classifier) : Real
DIT() : Real
DIT(c : Classifier) : Real
MinRFC(c : Classifier) : Real
AvgRFC(c : Classifier) : Real
MaxRFC(c : Classifier) : Real
MinCBO(c : Classifier) : Real
AvgCBO(c : Classifier) : Real
MaxCBO(c : Classifier) : Real
MinLCOM(c : Classifier) : Real
MaxLCOM(c : Classifier) : Real

**Fig. 1.** Extension to the UML 2.0 metamodel. *This UML package diagram shows the definition of the CK metrics as a separate package, with a dependency on classes from the UML metamodel.*

## 4 The Chidamber and Kemerer Metrics Suite

In this section we use the Chidamber and Kemerer (CK) [3] metrics suite to illustrate the use of the approach outlined in this report (see Figure 1). We express the CK metrics as OCL queries over the part of the UML 2.0 metamodel [17] that defines class diagrams. We are the first authors to provide such definitions using the UML 2.0 metamodel.

The UML 2.0 metamodel is a model that is used to define the UML. It specifies the constructs that may be used in a UML model and the relationships between these constructs. For example, the part of the UML metamodel that is specific to class diagrams defines the concepts of class, attribute, operation and states that a class includes attributes and operations. The structure of a UML model always conforms to the UML metamodel.

The metrics suite proposed by Chidamber and Kemerer is one of the most well known suite of object-oriented metrics. The suite consists of the following six metrics:

- Weighted methods per class (WMC)
- Depth of inheritance tree (DIT)
- Number of children (NOC)
- Coupling between object classes (CBO)
- Response for a class (RFC)
- Lack of cohesion in methods (LCOM)

The CK metrics were proposed to capture different aspects of an object-oriented design. However, not all of the CK metrics can be precisely measured from a UML class diagram. Implementation details, such as the code in the bodies of method definitions, are required to measure the CBO, RFC and LCOM metrics. However, we were able to provide definitions to estimate the values for these metrics based on the information in the UML class diagrams. Such measures are useful as they can provide upper and lower bounds for metrics calculated at later stages in the design or implementation process.

As an example of the format of the CK metric definitions, Figure 2 illustrates how the NOC metric can be expressed as an OCL query over the UML 2.0 metamodel. Here, the definition is parameterised by a single `Classifier`, and the body of the definition returns the size of the set of all children of this classifier. The auxiliary operation `children` traverses the elements and relationships in the UML metamodel to assemble this set. Full details of this and other metric definitions can be found in Appendix A at the end of this report.

```
-- Returns a count of all immediate descendants of the Classifier c
context ckmetricset::NOC(c:UML::Classifier):Real
body: self.children(c)->size()

-- Returns the set of all immediate descendants of the Classifier c
def: children(c:UML::Classifier):Set(UML::Classifier)
= self.scope->excluding(c)
            ->select(i:UML::Classifier| i.parents()->includes(c))
            ->asSet()
```

**Fig. 2.** NOC Metric Definition. *This OCL code defines the NOC metrics from the CK metrics suite, and is part of a larger definition of the whole CK metric suite which we have implemented using dMML.*

As a proof of concept, our tool dMML has been used to calculate these metrics for an open source project, *Velocity* which is part of the Apache Jakarta project [21]. We chose to use version 1.2 of *Velocity* as this is the version used in the study by Briand et al. [22]. We reverse engineered the system using Rational Rose to obtain a UML class diagram. Using our dMML tool we calculated the CK metrics suite for the resulting UML class diagram.

## 5 Summary and Future Work

In this report, we have identified the need for a clear, unambiguous framework for defining metrics. This framework should provide for the comparison of different definitions of the same metrics, and for using a metric, or suite of metrics in different environments. To achieve this we exploit OCL as a specification language, and harness the UML metamodel to provide a framework for metric definitions. We have implemented a tool, dMML, as an initial demonstration of the feasibility of our approach. A final contribution of this work is that it provides a first ever definition of the Chidamber and Kemerer metrics suite using the UML 2.0 metamodel as a basis for these definitions.

While our approach to date is similar to other research in this area, particularly that of Baroni et al., it differs in a number of key areas. Our approach decouples the metrics from the underlying metamodel. While this does not provide any immediate benefit for the specification of metrics over UML class diagrams, it is key to providing a foundation for our future work. First, our approach can be generalised at the metamodel level, for example, to apply to other UML diagrams. Second, the metric definitions and their calculation procedure is highly extensible, allowing for different versions to be implemented and compared.

We plan to build on this foundation by developing our research in three main directions:

– We plan to extend metric definitions to other UML diagrams. While this will allow us to add breadth to our metric set, it will also be important in ensuring consistency across design documents for a single application, and in tracking the impact of design decisions from different diagrams on the application as a whole.
– We will extend the metrics to the implementation level, using programming language metamodels. This will provide a single, coherent framework within which the design and implementation process can be measured. This will provide a clear, quantitative measure of the changes that take place between design and implementation.
– We will investigate the variances between different definitions of the same metrics over both design and implementation artifacts.

We have already tested the feasibility of our approach on the Jakarta *Velocity* tool. We intend to analyse a suite of open-source software as part of our work, in order to ensure the robustness and generalisability of our results.

## References

1. Basili, V., Briand, L., Melo, W.L.: A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering **22** (1996) 751–761
2. Li, W., Henry, S.: Object-oriented metrics that predict maintainability. Journal of Systems and Software **23** (1993) 111–122
3. Chidamber, S., Kemerer, C.: A metrics suite for object oriented design. IEEE Transactions on Software Engineering **20** (1994) 476–493
4. Lorenz, M., Kidd, J.: Object-Oriented Software Metrics. Prentice Hall Object-Oriented Series (1994)
5. Fenton, N., Lawrence Pfleeger, S.: Software Metrics: A Rigorous and Practical Approach. International Thompson Computer Press (1996)
6. Briand, L.C., Daly, J.W., Wuest, J.K.: A unified framework for coupling measurement in object-oriented systems. IEEE Transactions on Software Engineering **25** (1999) 91–121

7. Churcher, N., Shepperd, M.: Comments on 'A metrics suite for object-oriented design'. IEEE Transactions on Software Engineering **21** (1995) 263–265
8. Briand, L.C., Daly, J.W., Wuest, J.K.: A unified framework for cohesion measurement in object-oriented systems. Empirical Software Engineering **3** (1998) 65–117
9. Reißing, R.: Towards a model for object-oriented design measurement. In: Proceedings of ECOOP Workshop on Quantative Approaches in Object-Oriented Software Engineering, Budapest, Hungary (2001)
10. El-Wakil, M., El-Bastawisi, A., Riad, M., Fahmy, A.: A novel approach to formalize object-oriented design metrics. In: Proceedings of Evaluation and Assessment in Software Engineering, Keele, UK (2005)
11. Wilkie, F., Harmer, T.: Tool support for measuring complexity in heterogeneous object-oriented software. In: Proceedings of IEEE International Conference on Software Maintenance, Montréal, Canada (2002)
12. Baroni, A.: Formal definition of object-oriented design metrics. Master's thesis, Vrije Universiteit Brussel - Belgium, in collaboration with Ecole des Mines de Nantes - France and Universidade Nova de Lisboa - Portugal (2002)
13. Baroni, A., Braz, S., Brito e Abreu, F.: Using OCL to formalize object-oriented design metrics definitions. In: Proceedings of ECOOP Workshop on Quantative Approaches in Object-Oriented Software Engineering, Malaga, Spain (2002)
14. Baroni, A., Brito e Abreu, F.: A formal library for aiding metrics extraction. In: Proceedings of ECOOP Workshop on Object-Oriented Re-Engineering, Darmstadt, Germany (2003)
15. The Object Management Group: UML 1.3 specification (1999)
16. Goulão, M., Brito e Abreu, F.: Formalizing metrics for COTS. In: Proceddings of the ICSE Workshop on Models and Processes for the Evaluation of COTS Components, Edinburgh, Scotland (2004)
17. The Object Management Group: UML 2.0 draft superstructure specification (2003)
18. OMG: Meta Object Facility (MOF) Core Specification v2.0. Ref.: formal/06-01-01. Object Management Group (2006)
19. Warmer, J., Kleppe, A., Bast, W.: MDA Explained: The Model Driven ArchitecturePractice and Promise. Addison-Wesley (2003)
20. Warmer, J., Kleppe, A.: The Object Constraint Language. Addison-Wesley (2003)
21. Jakarta: The Apache Jakarta Project. http://jakarta.apache.org/ (2003)
22. Arisholm, E., Briand, L., Fyen, A.: Dynamic coupling measurement for object-oriented software. Technical Report 2003-05, Simula Research Laboratory, Norway (2003)

## A Definitions of the Chidamber and Kemerer metrics

```
/*
 * The following are OCL expressions
 * regarding Metrics::ckmetricset.
 */

package Metrics

-- Definition of additional Operations

context ckmetricset

-- Returns the set of methods implemented in the Classifier c,
excludes all non overriding inherited and abstract methods
def: implementedMethods(c:UML::Classifier):Set(UML::Operation)
= self.methods(c)-self.abstractMethods(c)

-- Returns the set of methods implemented in the Classifier c
def: methods(c:UML::Classifier):Set(UML::Operation)
= c.ownedElement->select(e:UML::Element|self.isKindOfMethod(e))
                ->collect(i:UML::Element|
                          i.oclAsType(UML::Operation))
                ->asSet()

-- Returns the set of abstract methods of the Classifier c
def: abstractMethods(c:UML::Classifier):Set(UML::Operation)
= self.methods(c)->select(m:Kernel::Operation|m.isAbstract())

-- Checks if an Element is a Method/Operation
def: isKindOfMethod(e:UML::Element):Boolean
= e.oclAsType(UML::Operation).oclIsUndefined()

-- Returns the maximum element in a set of integers
def: max(s:Set(Real)):Real
= s->iterate(elem:Integer; result:Integer = -1|result.max(elem))

-- Returns the set of immediate descendents of the Classifier c
def: children(c:UML::Classifier):Set(UML::Classifier)
= self.scope->excluding(c)->select(i:UML::Classifier|
                                   i.parents()->includes(c))
                          ->asSet()
```

```
-- Returns a set containing the immediate ancestors
-- of the Classifier c
def: parents(c:UML::Classifier):Set(UML::Classifier)
= self.scope->intersection(c.parents())

-- Returns the minimum set of all Classifiers that are
-- potentially coupled to the Classifier c
def: minCouplings(c:UML::Classifier):Set(UML::Classifier)
= self.scope->excluding(c)
            ->select(elem:UML::Classifier|
                     self.minCoupledTo(elem,c) or
                     self.minCoupledTo(c,elem))

-- Returns true if the Classifier x is coupled to Classifier y
def: minCoupledTo(x:UML::Classifier, y:UML::Classifier):Boolean
= self.hasDependency(x, y) or self.hasAttribute(x, y)

-- Returns true if the Classifier x has a dependency with Classifier y
def: hasDependency(x:UML::Classifier, y:UML::Classifier):Boolean
= self.dependencies(x)->includes(y)

-- Returns the set of suppliers of the dependency relationships
-- of the Classifier c
def: dependencies(c:UML::Classifier):Set(UML::NamedElement)
= c.clientDependency.supplier->asSet()

-- Returns true if the classifier x has an
-- accessible (within x) attribute of type y
def: hasAttribute(x:UML::Classifier, y:UML::Classifier):Boolean
= self.typesOfAllAccessibleAttributes(x)->includes(y)

-- Returns the types (Classifiers) of all the attributes of the
Classifier c that is accesible from within c
def: typesOfAllAccessibleAttributes(c:UML::Classifier)
    :Set(UML::Classifier)
= self.allAccessibleAttributes(c)->collect(q:UML::Property|q.type)
                                 ->collect(r:UML::Type|
                                   r.oclAsType(UML::Classifier))
                                 ->asSet()

-- Returns the set of all attributes
-- (including public and protected attributes of all parents)
-- of the Classifier c that are accessible within c
def: allAccessibleAttributes(c:UML::Classifier):Set(UML::Property)
= c.attribute->union(c.allParents()
             ->collect(i:UML::Classifier|i.attribute)->asSet()
             ->select(p:UML::Property|
                      p.getVisibility()=UML::VisibilityKind::public
                      or
                      p.getVisibility()=UML::VisibilityKind::protected))
             ->asSet()

-- Returns an estimation of the set of classes that
-- are coupled to the Classifier c
def: avgCouplings(c:UML::Classifier):Set(UML::Classifier)
= self.scope->excluding(c)
            ->select(elem:UML::Classifier|
                     self.avgCoupledTo(elem,c)
                     or
                     self.avgCoupledTo(c,elem))

-- Returns true if the classifier x is coupled to classifier y
def: avgCoupledTo(x:UML::Classifier, y:UML::Classifier):Boolean
= self.hasDependency(x, y) or self.hasAttribute(x, y) or
  self.hasParameter(x, y) or self.hasAssociation(x, y)

-- Returns true if at least one implemented method of
-- classifier x has a parameter of type y
def: hasParameter(x:UML::Classifier, y:UML::Classifier):Boolean
= self.typesOfParameters(x)->includes(y)

-- Returns the types of all the parameters of the
-- implemented methods of the Classifier c
def: typesOfParameters(c:UML::Classifier):Set(UML::Classifier)
= self.parameters(c)->collect(q:UML::Parameter|q.type)
                    ->collect(r:UML::Type|
                              r.oclAsType(UML::Classifier))
                    ->asSet()

-- Returns all parameters of all the implemented methods
-- of the Classifier c
def: parameters(c:UML::Classifier):Set(UML::Parameter)
= self.implementedMethods(c)->collect(o:UML::Operation|
                                      o.ownedParameter)
                            ->asSet()

-- Returns true if the Classifier x has an association
-- with Classifier y
def: hasAssociation(x:UML::Classifier, y:UML::Classifier):Boolean
= self.typesOfAssociations(x)->includes(y)
```

```
-- Returns the set of all Classifiers that have an association
-- relationship with the Classifier c
def: typesOfAssociations(c:UML::Classifier):Set(UML::Classifier)
= self.associations(c)->collect(q:UML::Association|q.endType)
                      ->flatten()
                      ->collect(r:UML::Type|
                                r.oclAsType(UML::Classifier))
                      ->asSet()

-- Returns the set of Associations for the Classifier c
def: associations(c:UML::Classifier):Set(UML::Association)
= c.ownedElement->select(e:UML::Element|isKindOfAssociation(e))
                ->collect(i:UML::Element|
                          i.oclAsType(UML::Association))
                ->asSet()

-- Checks if an Element is an Association
def: isKindOfAssociation(e:UML::Element):Boolean
= e.oclAsType(UML::Association).oclIsUndefined()

-- Returns a set containing all methods of all classes (except c)
-- within the scope that are accessible from all classes
def: maxMethodsCalled(c:UML::Classifier):Set(UML::Operation)
= self.scope->excluding(c)
            ->collect(i:UML::Classifier|self.publicMethods(i))
            ->union(self.allAccessibleInheritedMethods(c))
            ->asSet()

-- Returns a set containing an estimate of the minimum operations
-- called by the Classifier c
def: minMethodsCalled(c:UML::Classifier):Set(UML::Operation)
= self.minCouplings(c)->collect(i:UML::Classifier|
                                self.publicMethods(i))
                      ->union(self.allAccessibleInheritedMethods(c))
                      ->asSet()

-- Returns a set containing an estimate of the operations
-- called by the Classifier c
def: avgMethodsCalled(c:UML::Classifier):Set(UML::Operation)
= self.avgCouplings(c)->collect(i:UML::Classifier|
                                self.publicMethods(i))
                      ->union(self.allAccessibleInheritedMethods(c))
                      ->asSet()
```

```
-- Returns all public implemented methods of the Classifier c
def: publicMethods(c:UML::Classifier):Set(UML::Operation)
= self.implementedMethods(c)
       ->select(o:UML::Operation|
                o.getVisibility()=UML::VisibilityKind::public)

--Returns all inherited methods of the Classifier c that can
-- be called within c, this is the public and protected methods of
-- all parents of c
def: allAccessibleInheritedMethods(c:UML::Classifier)
     :Set(UML::Operation)
= c.allParents()->collect(i:UML::Classifier|
                          self.implementedMethods(i))
                ->flatten()->asSet()
                ->select(o:UML::Operation|
                         o.getVisibility()=UML::VisibilityKind::public
                         or
                         o.getVisibility()=UML::VisibilityKind::protected)

--Computes the sum of 1...n, where n is positive
def:sum(n:Integer):Real
= (n*(n+1))/2


--Metric Operations

-- Returns a count of all the operations of the Classifier c,
-- including all inherited operations
context ckmetricset::WMC(c:UML::Classifier):Real
body: self.implementedMethods(c)->size()

-- Returns a count of all immediate descendants
-- of the Classifier c
context ckmetricset::NOC(c:UML::Classifier):Real
body: self.children(c)->size()

-- Computes the DIT for the Classifier c
context ckmetricset::DIT(c:UML::Classifier):Real
body: if self.parents(c)->size() = 0 then -- c is the root
         0
      else --DIT for c is maximum DIT value of its parents
         self.max(self.parents(c)
             ->collect(i:UML::Classifier|self.DIT(i)+1)
             ->asSet())
      endif
```

```
-- Computes the DIT for the entire model
context ckmetricset::DIT():Real
body: self.max(self.scope->collect(c:UML::Classifier|self.DIT(c))
                         ->asSet())

-- Returns the maximum CBO of the Classifier c,
which is a count of all the Classifiers within the scope
context ckmetricset::MaxCBO(c:UML::Classifier):Real
body: self.scope->size()-1

-- Returns the minimum CBO of the Classifier c
context ckmetricset::MinCBO(c:UML::Classifier):Real
body: self.minCouplings(c)->size()

-- Returns an estimate of the CBO of the Classifier c
context ckmetricset::AvgCBO(c:UML::Classifier):Real
body: self.avgCouplings(c)->size()

-- Computes the maximum possible RFC value for the Classifier c,
-- which is a count of all accessible operations within the scope
context ckmetricset::MaxRFC(c:UML::Classifier):Real
body:self.implementedMethods(c)
     ->union(self.maxMethodsCalled(c))->size()

-- Computes the minimum possible RFC value for c
context ckmetricset::MinRFC(c:UML::Classifier):Real
body: self.implementedMethods(c)
      ->union(self.minMethodsCalled(c))->size()

-- Computes an average value for RFC for c
context ckmetricset::AvgRFC(c:UML::Classifier):Real
body:self.implementedMethods(c)
     ->union(self.avgMethodsCalled(c))->size()

-- Returns the maximum value for the LCOM of the Classifier c
context ckmetricset::MaxLCOM(c:UML::Classifier):Real
body: sum(self.implementedMethods(c)->size()-1)

-- Returns the minimum value for the LCOM of the Classifier c
context ckmetricset::MinLCOM(c:UML::Classifier):Real
body:  0

endpackage --Metrics
```