Assessing the Quality of your Software with MoQam

Jannik Laval, Alexandre Bergel, Stéphane Ducasse
RMoD Team, INRIA, Lille, France
firstname.lastname@inria.fr

1. INTRODUCTION

Over the last decade, the need for quality in software has increased. Several quality models have been proposed [3, 5, 9]. These models emphasize the need to have quality checks while developing a software program. As far as we are aware of, no model to assess quality of existing software have reached a significant acceptance.

This paper describes the Qualixo quality model. Qualixo is an open-source quality model developed by several companies and pushed further in the context of the Squale research project. According to Marinescu and Ratiu [?], Qualixo can be classified as a Factor-Criteria-Metrics quality model. Qualixo is being applied in large companies such as AirFrance or PSA. It uses measurements to assess software quality. These measurements cover a number of different aspects of a software, including specification accuracy, programming rules, and test coverage. Qualixo has been originally implemented on top of Eclipse. In this paper we present MoQam (Moose Quality Assessment Model), the implementation of the Qualixo quality model in the Moose open-source reengineering environment.

This paper is organized as follows: Section 2 presents the model and its four components. Section 3 describes how the model is implemented in Moose. Section 4 offers two examples of notation and a real test of MoQam. Section 5 gives a brief overview of the related works, and finally, Section 6 concludes.

2. OVERVIEW OF THE QUALIXO MODEL

The model defined by Qualixo is composed of four elements, each having a different granularity. Figure 1 presents the four levels of the Qualixo model. Starting from the most fine-grained element it is composed of:

- a *metric* is a measurement directly computed from the program source code. The current implementation provides 17 metrics.
- a *practice* assesses one quality of a model. A practice is associated to one or more metrics. 50 practices are currently implemented.
- a *criteria* assesses one principle of software quality by weighting a set of practices. Such a principle could be safety, simplicity, or modularity. For now, 15 criteria are implemented.
- a *factor* represents the highest quality assessment. A factor is computed over a set of weighted criteria. 6 factors are currently available and are explained in a following section.

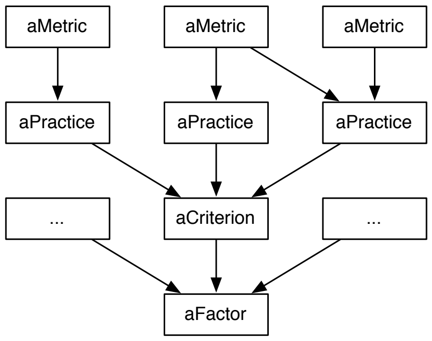

Figure 1: Meta-model of Qualixo Model.

Metrics. A metric is a measurement on a source code. it is a function that returns a numerical value which evaluate a source code. As such, metrics are considered as low level. Literature on metrics provides a significant amount of metrics such as the well known *Depth Inheritance Tree* (DIT, INH) and *Specialization Index* (SIX) [4,6,7]. Figure 2 shows the different information collected to fully represent a metric. It is composed by an acronym, a name and a list of alternative names (a metric may have several name), a formulae referencing a function object, a version number (the version number refers to possible different implementations of the same metrics), a reference pointing to the articles precisely defining the metrics, possible metrics that could be used to replace the current one, a reference source and their associated results against which the metric can be tested for non-regression, supportedBy lists the list of tools implementing the metrics.

Practices. A practice is a composition of metrics that weights and scales the metric's value down to make it range between 0 and 3. By convention, 3 is a good mark meaning that the software under analyses does it well for the given metric. A practice may be associated to one or more metrics.

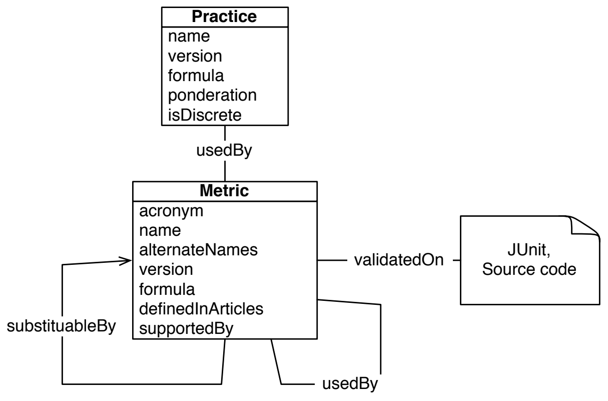

Figure 2: Meta model for metrics.

The practice formula is defined in two different ways: either by a continuous function, or by a discrete function using metric thresholds. For example, a discrete function is used in practice *Inheritance Size*. It is based on *Depth Inheritance Tree* (DIT) [7].

```
Value is 0 if DIT > 7
Value is 1 if SIX > 6
Value is 2 if SIX > 5
Value is 3 if SIX  =< 5
```

A practice is a formula based on metrics. Most of the time, the formula weights different metrics it is combining to emphasize a particular characteristic of the software under analysis. A practice may be discrete or not. A discrete function is used by certain practices to highlight component which are badly noted. It imposes thresholds which should not be exceeded whereas a continuous function is linear.

As an illustration, consider the practice *Coupling Class Efferent*. It is based on the *Coupling Between Object* (CBO) metric [1]. This formula returns a value between 0 and 3:

```
FAMIXClass>>notationOfComponentEfferentCoupling
   | result |
   result:= ((10- (self sureReferencedClasses size))/3) exp.
   (result < 0) ifTrue: [^ 0].
   (result > 3) ifTrue: [^ 3].
   ^ result
```

Another interesting practice is *Documentation*. In our implementation, this practice is associated with FAMIXMethod entity since it is based on metrics *Number of Lines of Code* which counts the lines of codes (and is equivalent to LOC) and *Number Of Comments* which counts the number of lines of comments.

Quality Criteria. Practices are gathered and balanced to define criteria values. A criteria groups practices by categories. For example, the criteria *level of interdependence* groups practices *efferent coupling class* and *related coupling class* because these two practices show dependencies between classes.

Quality Factor. A factor gives a mark, called a quality factor, between 0 and 3 for an average of criteria. Note that the criteria used by a factor must be coherent and, for example, measures the same entities. A factor is the highest level of evaluation. Therefore, a factory quality is computed for the whole program and may potentially covers all elements of the meta-model.

Six quality factors are currently defined in Qualixo:

- *Functional capacity* represents the adequacy between the needs and functionalities offered.
- *Architecture* corresponds to the technical architecture quality.
- *Maintainability* represents the facility to correct residual errors.
- *Capacity to evolve* measures the capacity to add functionalities.
- *Capacity to re-use* represents faculty to re-use the code artifacts.
- *Reliability* represents the stability of the program.

Thus, a complete assessment of a software system results in 6 quality factor. A quality factor below 1 is interpreted as a failure in meeting its quality objective. Between 1 and 2, the quality objective is considered as achieved with reserve. Above 2, the objective is considered as achieved.

Applying the Qualixo model on a given software system cannot be fully automated. Some metrics cannot be automatically extracted from the code, and thus require human intervention. Typically, metrics related to the documentation fall into this category.

3. IMPLEMENTATION

MoQam is the partial implementation of the Qualixo model on top of the Moose reengineering environment [2, 8] based on the FAMIX source code meta-model (See Figure 3). Only metrics that can be automatically computed are covered by our implementation. Therefore, practices, criteria and factors related to these software metrics are provided by our implementation (9 practices, 4 criteria, a part of 2 factors).

Our implementation enriches FAMIX element classes: class related metrics and practices are implemented as extension in FAMIXClass, method related metrics and practices in FAMIXMethod, package related metrics and practices in FAMIXPackage.

Moose comes with a large set of metrics. MoQam benefits from them by mapping each MoQam metric into a Moose one. As factors and criteria are application level aggregates, they are implemented in the class MooseModel.

As an example, consider the practice *Inheritance Size*. It aggregates one metric: DIT, which is weighted by a discrete function as shown in ncInheritance, and then is weighted by npInheritance. The practice does an average of the weighting function and returns a result between 0 and 3. The implementation is the following:

```
FAMIXClass>>DIT
  ^ self superclassHierarchy size
```

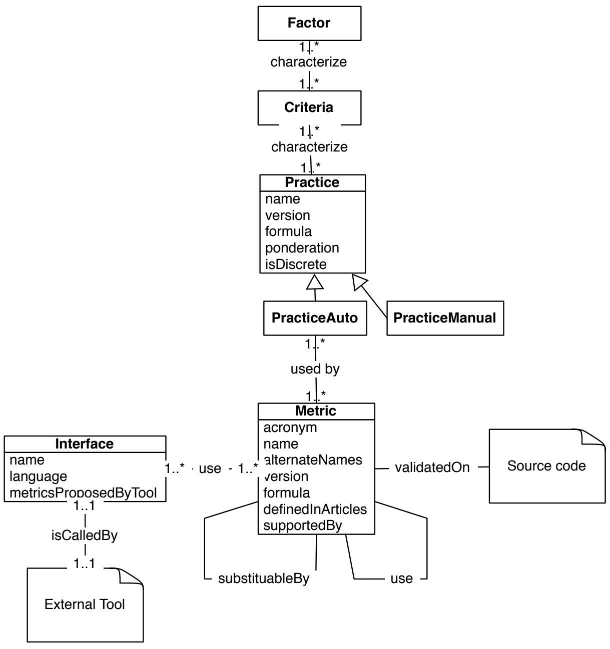

Figure 3: Qualixo meta-model implementation in FAMIX.

```
FAMIXClass>>notationOfComponentInheritance
   (self dit > 7) ifTrue:[^ 0].
   (self dit > 6) ifTrue:[^ 1].
   (self dit > 5) ifTrue:[^ 2].
   ^  3

FAMIXClass>>notationOfPracticepInheritance
  ^  (20  ** ((self notationOfComponentInheritance) negated))

MooseModel>>practiceInheritanceSize
  | practice result |
  practice:= self allClasses inject: 0 into:
              [:first:each | first + each notationOfPracticepInheritance ].
 result:= practice / self allClasses size.
 ^ ((result log) / (20 log)negated)
```

4. CASE STUDIES

We apply the Qualixo model to two large software systems: Moose (170 Classes) and ArgoUML (1654 Classes). Yet, just some practices and criteria are implemented.

Moose. This is a test of MoQam Implementation on the package Moose. Just some practices are tested here:

```
result for practiceAfferentCoupling 1.47
result for practiceEfferentCoupling 1.04
result for practiceDocumentation 0
result for practiceInheritanceSize 0.83
result for practiceMethodsNumber: 3
result for practiceMethodsSize: 0
result for practiceSpaghettiCode: 0
```

Here, we can say:

- Links between classes are acceptable (practiceAfferentCoupling is equal to 1.47 and practiceEfferentCoupling is equal to 1.04)
- There is little or no documentation (practiceDocumentation) in methods.
- The depth inheritance tree is bad (practiceInheritanceSize less than 1). When we look the inheritance tree of MooseCore, we can notice that some classes have a hierarchy size higher than 6 (like all classes which inherit from FAMIXAbstractNamedEntity). The *DIT* of the class FAMIXAbstractNamedEntity is 5, because *DIT* is calculated from the class Object. So this is clearly an indication that the metric should be interpreted since we know that the inheritance hierarchy of Moose and in particular the FAMIX model is good and optimal.
- The number of methods for a class is good (practiceMethodsNumber).
- But the size of methods and its complexity is high (practiceMethodsSize and practiceSpaghettiCode).

Here, results show mainly that this is less documentation and it is necessary to add this. The result

ArgoUML. This is a test of MoQam Implementation on ArgoUML. Just some practice are tested here:

```
result for practiceAfferentCoupling: 3.0
result for practiceEfferentCoupling: 3.0
result for practiceDocumentation: 0
result for practiceInheritanceSize: 0.3541
result for practiceMethodsNumber: 0.0012
result for practiceMethodsSize: 0
result for practiceSpaghettiCode: 0
```

We can say that just links between classes are good. The rest has bad notation : there is no or little documentation, the inheritance tree is high, the number, the size and the complexity of methods is high.

Then criteria can be computed: the criterion *criteriaInterdependanceLevel* is an average of *practiceAfferentCoupling* and *practiceEfferentCoupling*. Thus its result is 3.0, the level of independence of ArgoUML is very good. And the criterion *criteriaSimplicity* is an average of *practiceSpaghettiCode*, *practiceMethodsNumber* and *PracticeMethodSize*. Thus its result is 0, the simplicity of ArgoUML is very bad.

This list shows that we can improve the software. According to the results, things to do is : first, to add more documentation in methods and to review the number and the size of methods and third to analysis the inheritance tree because it is high.

This analysis can be used by managers or engineers to improve the process of reengineering of a software.

5. RELATED WORK

Swat4j. Swat4j[1] is a quality model for Java. It comes with about 30+ Metrics and about 100+ Industry Standard Best Practice Rules. Swat4j is designed based on the principles of ISO 9126-1 (Quality Model) and ISO 9126-3 (Software Product Quality, Internal Metrics).

[1]http://www.codeswat.com

Hierarchical model. The Quality Model for Object-Oriented Design (QMOOD) model in use has lower-level design metrics defined in terms of design characteristics, and quality is assessed as an aggregation of the model's individual high-level quality attributes. These high-level attributes are assessed using a set of empirically identified and weighted object-oriented design properties [**?**].

QMOOD involves four levels (L_1 through L_4), and three mappings (L_{12}, L_{23}, L_{34}) used to connect the four levels. While defining the levels involves identifying design quality attributes, quality carrying design properties, object-oriented design metrics, and object-oriented design components, defining the mapping involves connecting adjacent levels by relating a lower level to the next higher level.

QMOOD assesses for each component its quality. Squale provides globals appreciations in terms of factors.

Detection strategies. Marinescu and Ratiu [?] raised the following question *How should we deal with measurement results?* After having pinpointed few limitations in Factor-Criteria-Metric models (e.g., obscure mapping of quality criteria onto metrics, poor capacity to map quality problems to causes), they introduce detection strategies as a generic mechanism for analyzing a source code model using metrics. The use of metrics in the detection strategies is based on mechanisms of filtering and composition. A filtering operation is characterized with thresholds and extremities. Composition operators are and, or, butnotin.

Based on the detection strategy mechanism, a new quality model is proposed, called *Factor-Strategy*. This model uses a decompositional approach, but after decomposing quality in factors, these factors are not anymore associated directly with a bunch of numbers. Instead, quality factors are now expressed and evaluated in terms of detection strategies, which are the quantified expressions of the good-style design rules for the object-oriented paradigm.

6. CONCLUSION

This paper describes MoQam, an implementation of a quality model for software named Qualixo. An implementation is also described and two case studies have been realized. Results of these case studies are encouraging to continue to develop it.

This approach has three positive points. First, MoQam uses four levels of granularity, which offers different scopes for an analysis. Second, the weighting to calculate the practices allow bad coding style to be highlighted. Third, these results may be used in a dashboard.

Future work will focus on four points: first, we will weight practices to calculate criteria. That allows the same purpose that the weighting of practices: we will be able to highlight bad programming style. Second, we will integrate some metrics to compare and replace metrics which can be debatable (like *LCOM*, which can be replaced by *TCC and LCC* (Tight and Loose Class Cohesion), for example). The third thing is to integrate the manual practices of the model to have all factors functional. We project to experiment this model on several package which we wish to study. The Fourth point is to allow to personalize criteria (which practices used) and factors (which criteria used) according to project.

7. REFERENCES

[1] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[2] S. Ducasse, T. Gîrba, M. Lanza, and S. Demeyer. Moose: a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005.

[3] R. L. Glass. *Building Quality Software*. Prentice-Hall, 1997.

[4] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

[5] H.-W. Jung, S.-G. Kim, and C.-S. Chung. Measuring software product quality: A survey of iso/iec 9126. *IEEE Softw.*, 21(5):88–92, 2004.

[6] S. H. Kan. *Metrics and Models in Software Quality Engineering*. Addison Wesley, 2002.

[7] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.

[8] O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[9] D. Spinellis. *Code Reading The Open Source Perspective*. Addison-Wesley, 2003.