JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 4, No. 4, May-June 2005

Automated Measurement of UML Models: an open toolset approach

Luigi Lavazza, CEFRIEL and Politecnico di Milano
Alberto Agostini, Link snc, Corsico

Abstract

The Unified Modeling Language (UML) is the de facto standard language for modeling object-oriented software systems. As the importance of UML within organizations increases, the need for measuring UML models arises. This paper describes a UML measurement tool that not only fully supports the measurement of models according to the most popular metrics definitions, but also provides an open measurement base supporting user-defined metrics, unforeseen analysis, and process measurement.

1 INTRODUCTION

Modeling is playing an increasingly important role in the development of object-oriented software. In particular UML is becoming extremely popular as a modeling language for object-oriented systems. In order to manage the development process it is of crucial importance to be able to derive accurate quantitative knowledge from the software artifacts. In particular, in the early stages of development the UML models should be measured in order to provide project managers with the information needed to take management decisions. Several object-oriented metrics proposed for code are applicable to models too (possibly with minor adjustments). Thus it is reasonable to expect that measurement of models anticipates some knowledge concerning the implementation phase. This is a very relevant issue, since management decisions could be based on data that are available earlier than code measures, and yet are both objective (as they are derived from the models according to well defined measurement rules) and reliable (as the models determine to a large extent how the system will be implemented). Since in industrial software development processes it is not viable to derive measures manually, the measurement process must be automated, in order to guarantee efficiency and reliability.

This paper describes a tool that automates the measurement of UML models, and derives the quantitative information needed for technical and managerial purposes. Note that here we consider only the issue of measuring models in an automatic way; discussing the usage of the automatically derived measures is out of the scope of the paper.

Cite this article as follows: Luigi Lavazza and Alberto Agostini: "Automated Measurement of UML Models an open toolset approach", in *Journal of Object Technology*, vol. 4, no. 4, May-June 2005, pp. 115-134. http://www.jot.fm/issues/issue_2005_05/article2

Requirements for a tool supporting the measurement of UML models

Based on our experience, a tool supporting the measurement of UML models should satisfy the following requirements.

- It should be possible to measure UML models regardless of the CASE tool used to edit them. Therefore, a first very practical requirement is that the measurement tool has to be independent form the CASE tool used to build the models. An effective way to pursue this objective is to rely on a unique file format directly or indirectly supported by all the UML CASE tools.
- There is no unique suite of metrics for object-oriented models which is generally recognized to provide a complete and satisfactory set of indicators. Actually, since measurement is a typical goal-oriented activity, and each developer or project manager may have different goals, probably there will never be such a universal UML metrics suite. Therefore it is important that a UML measurement tool be able to support the most widely known metrics suites as well as user-defined metrics.
- Very often a model is obtained by updating an existing model. In fact, the greatest part of the software development effort is devoted to the maintenance (or evolution) of existing software. In such cases, the project manager is generally interested in a quantitative evaluation of what has been added or changed, rather than in the absolute dimensions of the new model. Therefore, in order to effectively support the maintenance/evolution process, a UML measurement tool must be able not only to measure a single model, but also to measure the difference between two versions of the same model.
- Although of great importance, models account only for a fraction of the development process. In order to get a comprehensive insight into the process, the process owner usually needs to analyze the measures of different kinds of artifacts, including models as well as code, problem reports, etc. Therefore, a tool supporting the measurement of UML models should allow the user to combine easily UML metrics with other kinds of metrics.
- A final requirement of great practical importance concerns interoperability. In fact, the outcome of the tool should be easily usable by other tools. In particular, users will be interested in performing statistical elaborations on the measurement data, in providing input to estimation tools, in creating graphs and reports, etc. In conclusion, the measurement tool should yield results in a form that is easily usable.

The tool described in the rest of this paper satisfies all of the requirements described above.

Structure of the paper

Section 2 briefly illustrates the object-oriented metrics that have been proposed by various authors and constitute a basic set of metrics that should be supported by UML measurement tools.

Section 3 describes the conceptual architecture, design and implementation of our UML measurement tool.

Section 4 illustrates how our tool can be used to collect a few of the most commonly needed metrics.

Section 5 describes how our tool supports the measurement of variations among different versions of UML models, thus supporting the evaluation of maintenance and evolution activities.

Section 6 illustrates the application of the tool in an industrial development process.

Section 7 compares our work with other approaches to UML model measurement.

Finally, section 8 draws some conclusions and sketches future work.

2 OBJECT-ORIENTED METRICS

In the last decades the object-oriented paradigm has been largely employed in the software development process. The metrics previously defined for traditional software development techniques soon proved not applicable to the new paradigm: as a consequence, several new metrics suites were proposed to capture the characteristics of object oriented models and code. A quite comprehensive survey, analysis and explanation of the numerous object oriented metrics proposed in the literature can be found in [Purao03].

The metrics applicable to UML models are primarily those proposed for evaluating object-oriented designs. In fact, these metrics concern typical object-oriented features that are common to practically all the object-oriented notations, including UML. Probably the best known of these metrics is the suite proposed by Chidamber and Kemerer [Chidamber94]: it includes six object-oriented metrics overcoming the limitations of the more traditional metrics. They are:

- “Weighted Methods per Class (WMC)”: this is the number of methods per class, weighted according to their complexity. A class having a big WMC is expected to call for a big effort for development and maintenance, to have a big impact on subclasses, and to be difficult to reuse. Note that often all methods of a class are considered to be equally complex: in these cases the value of WMC represents the number of methods of the class.
- “Depth of Inheritance Tree (DIT)”: it is the distance of the class from the root of the inheritance hierarchy. A high DIT implies that a big number of properties are inherited, therefore the behavior of the class is largely affected by its ancestors, and it will be probably difficult to evaluate.
- “Number Of Children (NOC)”: it is the number of classes that inherit directly from the considered class. A big NOC indicates a large reuse of the class.
- “Coupling Between Object classes (CBO)”: it is the number of classes which the considered class is coupled with. “Coupling” generally means that a class depends on another class, e.g., because it uses the other’s properties. A high CBO indicates little modularization. The consequences are a higher sensibility of the class to changes in other classes, and more complex testing.

- "Response For a Class (RFC)": it is the number of methods (both internal and external) that can be invoked in response to a message directed to a class instance. It is a measure of the complexity of the class in terms of communications with other classes. It is expected that the difficulty of managing classes (in terms of development, debugging, etc.) is somehow proportional to the RFC.
- "Lack of COhesion in Methods (LCOM)". It is defined as the number of methods pairs that access non overlapping sets of properties minus the number of methods pairs that access overlapping sets of properties (the metric is set to zero whenever the above subtraction is negative). Cohesiveness of methods within a class is considered desirable, since it promotes encapsulation. Low cohesion means that a single class collects the properties that should be better located into two or more subclasses. This increases complexity, thus increasing the likelihood of errors during the development process.

Chidamber and Kemerer defined their suite of metrics for a generic object-oriented design framework, thus it is quite easy to redefine them in terms of elements of the UML (or, more specifically, in terms of the metamodel of UML, as discussed in section 3).

Chidamber and Kemerer's metrics are by far the most widely known and used, since they incorporate the most important concepts that can be adopted to derive object-oriented quality indicators. For instance, correlations of Chidamber and Kemerer's metrics to fault-proneness [Basili96] and managerial indicators [Chidamber98] have been studied. However, several other metrics have been proposed [Cartwright00, Kim02, Lorenz94]. In particular, the metrics proposed by Cartwright and Shepperd are interesting because they involve the measurement of UML state diagrams. In fact, they include the measure of the number of states per class and the number of events that affect the behavior of each class. In particular, Cartwright and Shepperd found a strong linear relationship between the size of a class in terms of LOC and the number of states per class. This is an interesting result because several prediction models use LOC size as the main independent variable.

We provided our tool with built-in support of a set of metrics which includes most of the Chidamber and Kemerer's metrics as well as Cartwright and Shepperd's metrics. However, this choice does not represent a commitment to these metrics, since the openness of the tool allows the user to easily extend the set of supported metrics (as discussed in section 3).

We decided to measure only class and state diagrams, because these are generally reported as the sources of the most relevant indications concerning the qualities of the object-oriented models. This must not be regarded as a big limitation, since it is possible –with a little of coding– to extend our approach to the measurement of the other diagrams of a UML model.

3 DESIGN AND IMPLEMENTATION OF A TOOL AUTOMATING THE MEASUREMENT OF UML MODELS

The definition of UML by the Object Management Group [OMG03] includes a precise definition of the metamodel of the language. The metamodel describes the syntax and –to some extent– the semantics of UML models. For instance, the UML metamodel defines the concepts of package, class, attribute, etc; it states that a class includes attributes and methods, that associations can be established between classes, etc. The structure of a given UML model always conforms to the metamodel: therefore a measurement tool has to extract from the model the information conforming to the metamodel and derive the required indicators.

In order to read the information contained in the model it is preferable to access a textual version of the model, so that it can be analyzed (and interpreted) according to the meta-model. Luckily, the OMG has established a standard for the representation of UML models. This standard, called XMI (for XML Metadata Interchange [OMG00]), specifies how to represent a UML model in a way that is both XML-compliant and conformant to the UML metamodel. XMI allows different tools to exchange UML models using a common format: in fact, most UML editors are now equipped with facilities to convert UML models from their proprietary format into XMI.

The availability of UML models in XMI format is a great opportunity for the purpose of measuring UML models: in fact XMI files contain all the relevant information to be measured, conforming to the metamodel, and independent from the tool used to generate the model.

Exploiting XMI not only allows us to comply with the first of our requirements (independence from the UML tool), but also eases the development of the tool. In fact we identified two rather straightforward ways for implementing our UML Model Measurement Tool (UMMT):

1. One possibility is to store the contents of the given XMI file into a XML database. Measures can then be obtained by querying the database. Native XML databases and XML query languages are appearing, thus making this approach viable.
2. An alternative approach exploits the fact that XMI is a “specialization” of XML. Being XML a widely popular standard format, several utilities to parse XML files are available. By means of such utilities it is very easy to build an abstract representation of the input file (i.e., of the given model). The abstract representation can then be visited as required in order to compute the target metrics.

We chose an implementation strategy that is a sort of trade-off between the two approaches described above. We decided not to use XML databases because of two main reasons: they are not yet a mature technology, and they do not guarantee a level of interoperability with other tools as do –for instance– relational databases. Instead, we

retained the idea of storing the relevant contents of a model into a database, and to perform measurements by means of suitable queries. The logical structure of the UMMT is illustrated in Fig. 1. The organization of the tool has several advantages:

1. Using XMI files as input makes the tool independent from the source of the models.
2. XMI files can be parsed very easily by means of the available utilities originally developed to deal with XML files.
3. The UML metrics most commonly used do not consider the whole given model: generally only the class and state diagrams are subject to measurement. Our tool performs a selection of the contents of the model, and stores them in a relational database. Dealing with a smaller amount of data makes the following measurement activities faster.
4. The schema of the database is defined very simply after the metamodel.
5. Measurement is performed by running queries on the contents of the database. We defined queries corresponding to the most common UML metrics: these queries are ready to be executed by the user. In case the user needs other kinds of metrics, he/she has simply to define the corresponding queries.
6. Measures can be stored in the database as well. Thus it is very simple to produce reports, to export data towards other tools, or to let other tools retrieve the needed data from the DB.

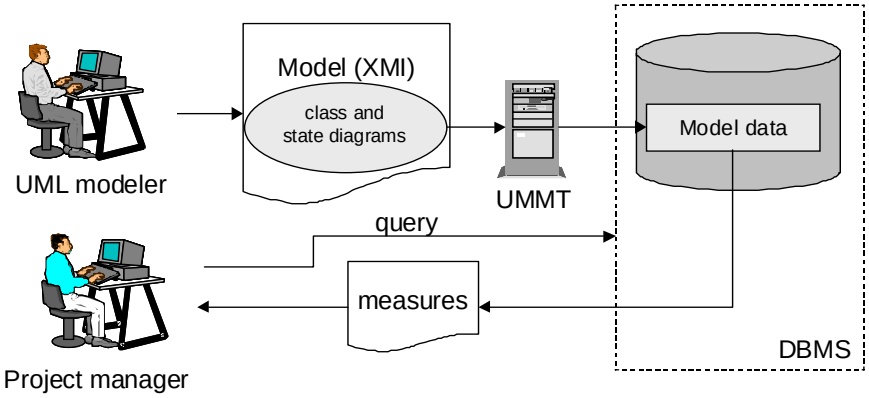


Fig. 1. The logical structure of the UML measurement tool.

In order to define new metrics the user has to know the schema of the database implementing the metamodel. This is not difficult, as the metamodel has been suitably simplified (see Fig. 2 and Fig. 3), and the conceptual schema of the database is a plain transposition of the metaschema itself. Given the complexity of UML, and in particular considering the number of diagrams contained in a UML model, it is possible that the user conceives metrics that our current implementation does not support, e.g., to count use cases. In such a case, the user should modify the UMMT itself in order to extract the required data. This is not a difficult task for people mastering XML.

Our approach requires that we define a (relational) database which is suitable for storing UML model data, i.e., data that comply with the definition of the UML metamodel. Therefore we designed a database having such features: the metamodel is the conceptual schema of the database, while the logical schema is obtained applying the usual normalizations, optimizations, etc. The only relevant limit of our database with respect to the ability to represent UML models is due to the fact that we implemented only a subset of the UML metamodel, since we are not interested in all the data contained in the source models. In particular, we tried to achieve a trade-off between efficiency and content: on one hand we wanted to feed the database with as little information as possible, on the other hand all the information required to compute the most common metrics should be present. As a result, we selected the following elements for inclusion in the simplified metamodel:

- packages, together with containment relations;
- classes, with the indication whether they are abstract or not, and the package they belong to;
- interfaces, together with the indication of the associated package or class;
- attributes of a class, with their type and visibility (public, protected or private);
- methods of a class, with their signature and visibility (public, protected or private);
- realization relations (as they are useful to specify how an interface is implemented);
- associations between classes and/or interfaces, with the indications whether they are aggregations or compositions;
- dependency relation between classes, interfaces and packages;
- generalizations relations.

From state diagrams we selected states and sub-states (together with their inclusion relations), transitions, and the associated triggering events, conditions and actions.

We specified the simplified metamodel by means of Entity/Relationship diagrams. Fig. 2 and Fig. 3 represent the metamodel concerning the structural part of UML (class diagrams) and the statecharts, respectively.

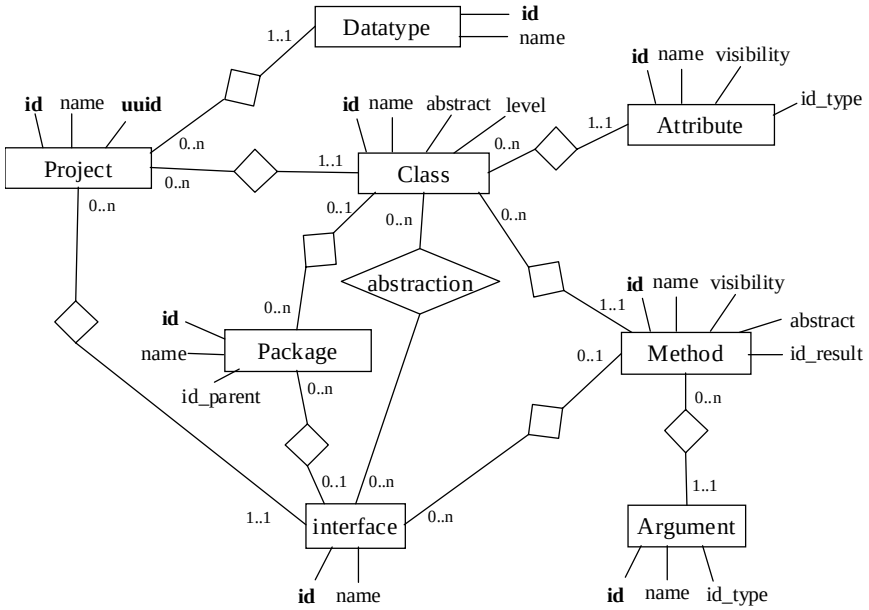


Fig. 2. Simplified metamodel of UML class diagrams.

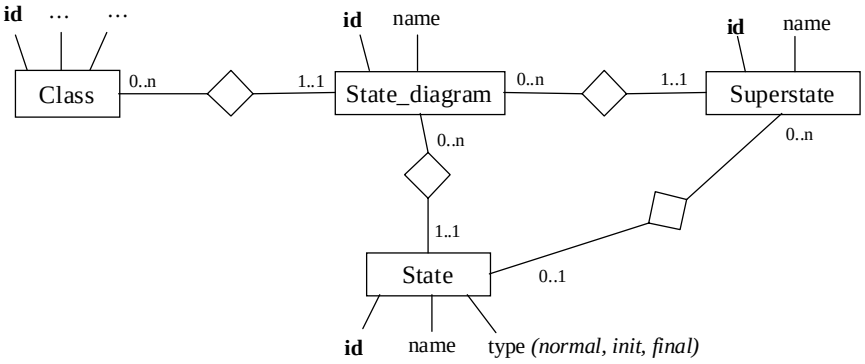


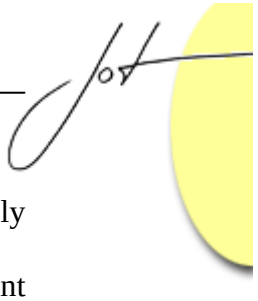
Fig. 3. Simplified metamodel of UML state diagrams.

Note that to make the figures readable we omitted a few elements that are actually present in our metamodel:

- in the structural part we have also associations, aggregations, compositions, as well as dependency relations (among classes, interfaces, etc.) and generalizations;
- in the statechart part we have also transitions, with the indication of triggering events, conditions, and associated actions.

The complete logical schema of the database is not reported here for space reasons. It can be found in [Agostini03].

The tool was implemented in Java, in order to assure portability to different environments. The Database Management System employed is MySQL [MySQL]. It was chosen because it is open source and sufficiently reliable and fast. The UML Model



Measurement Tool interfaces with the data-base by means of a module that can be easily reconfigured in order to use a different DBMS supporting standard SQL.

The main component of the tool is the one that reads XMI files, extracts the relevant information and populates the database with such information. The XMI parser is based on the well-known Xerces library for XML [Xerces].

4 PERFORMING MEASURES

As already mentioned, measures are obtained by querying the database. For instance, in order to get the number of classes contained in a model, one has to perform the following query:

```
SELECT count(*)
FROM Class
WHERE id_project=@prj
```

Although the logical schema of the database was not reported here, the reader can easily understand the details of the queries. For instance in the query above `id_project` is an attribute of table `Class` identifying the project which the class belongs to. `@prj` is a variable whose value must be set to the identifier (named `uuid`) of the project we are interested in.

The concept of project corresponds to the concept of model. We used the term "project" to stress that in a single database one can collect data from models belonging to different systems (or projects). The list of projects contained in the database can be obtained as follows:

```
SELECT name, uuid
FROM project
```

As an example of metrics belonging to the Chidamber & Kemerer suite, let us consider the Number Of Children (NOC) and the Depth of Inheritance Tree (DIT).

The NOC of the class identified by `@id_class` in a project identified by `@proj` is computed as follows:

```
SELECT count(*)
FROM generalization
WHERE id_project=@prj AND parent=@id_class
```

where generalization is a table that contains four attributes:

- `id:` the unique identifier of the relation;
- `descendant:` the identifier of the subclass in the generalization hierarchy;
- `parent:` the identifier of the superclass in the generalization hierarchy;
- `id_project:` the uuid of the project.

The DIT of the class identified by `@id_class` in a project identified by `@proj` is computed as follows (`level` is an attribute of table `Class` that represents the distance from the root class in a generalization hierarchy):

```
SELECT level
FROM Class
WHERE id=@id_class AND id_project=@prj
```

More complex metrics can be computed as well: for instance, the following query computes how many classes do not belong to a generalization hierarchy (in a given project):

```
SELECT count(*)
FROM  Class
WHERE id_project=@prj
      AND   level=0
      AND id NOT IN (SELECT parent
         FROM generalization
         WHERE id_project=@prj)
```

On the contrary, the following query computes how many classes are root of a generalization hierarchy (in a given project):

```
SELECT count(*)
FROM  Class
WHERE  id_project=@prj
       AND level=0
       AND id IN (SELECT parent
                  FROM generalization
                  WHERE id_project=@prj)
```

As an example of metrics concerning the state diagrams, consider the query that counts the number of states (excluding initial and final ones) of a class identified by `@id_class` in project `@prj`:

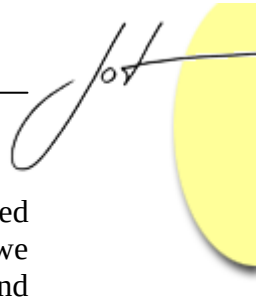
```
SELECT COUNT(*)
FROM  State s, State_diagram d
WHERE  d.id_class=@id_class AND s.id_diagram=d.id
AND s.type='normal' AND s.id_project=@prj AND d.id_project=@prj
```

5 MEASURING VARIATIONS OF UML MODELS

The importance of measuring variations of UML models can be understood by means of a very simple example. Suppose that version *n* of a given UML model contains 100 classes. Version *n+1* of the same model contains 110 classes. Since version *n+1* was obtained by modifying version *n*, the new measure does not provide a representation of the work done (and –worse– of the work still to be done in the following implementation and test phases). In fact, both of the following cases are possible:

a) in the new version, 10 new classes were introduced, while the former 100 remained unchanged;
b) the new version was obtained by dropping 20 classes, modifying other 25 classes, and introducing 30 new classes.

It is quite clear that in case b) we expect that the development effort will be much greater than in case a). Similarly we expect that more defects will be introduced, and therefore



more testing effort will be required. In order to deal with this kind of evolution-oriented development we need to measure the *differences* between the two versions. In fact, we expect that the development effort will be proportional to the number of new and modified classes (with possibly some contribution from the deleted classes too).

The measurement of differences is a feature of fundamental importance, since it can provide indicators that are essential to assess relevant features of the product or process. For instance, the variance of the changed (added or updated) classes in a specification model indicates the stability of requirements.

Given the structure of UMMT, the measurement of differences between model versions is achieved quite simply: we load the database with data coming from two different versions, making sure that every piece of data is associated with an identifier of the version it belongs to. Then it is quite straightforward to write queries that take into account version identifiers and measure differences. For instance it is possible to subtract the set of classes belonging to version *n* from the set belonging to version *n+1*, thus obtaining the set of new classes. The process is illustrated in FFig. 4.

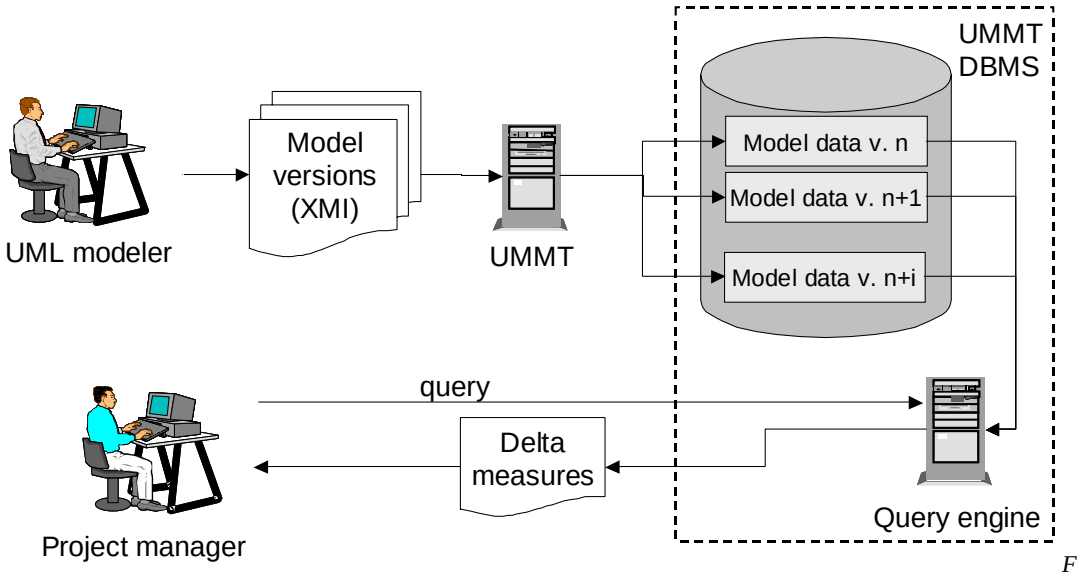


Fig. 4. Measurement of UML model evolution.

Technically, the idea is implemented as follows. We exploit an identifier, called uuid, which is present in every XMI file. For instance, the following XMI fragment specifies that the uuid of the model is “S.11”:

```
<XMI.content>
    <Model_Management.Model xmi.id="S.100050" xmi.uuid="S.11">
```

If the UML editor does not manage the uuid in an effective way, it is possible to edit the XMI file by means of a text editor and change the uuid. For instance, we could set `xmi.uuid="S.1.version2"`. The uuid is thus naturally part of the XMI file. We just need to load this piece of information in the database. Then we can set variables v1 and v2 with the uuids of the considered versions:

```
SET @v1='S.1'
SET @v2='S.1.version2'
```

Now it quite easy to find the classes that are in model v2 and not in v1:

```
SELECT name FROM Class
WHERE id_project=@v2
AND name NOT IN (SELECT name
                 FROM Class WHERE id_project=@v1)
```

Actually the former query finds the classes whose name appears in v2 and not in v1. In fact, whenever two elements of the model have to be compared, we need a criterion to perform the comparison. It is the user who has to decide whether simple criteria (like the comparison of names) are suitable, or more complex queries are needed. The tool supports metrics based on both simple and complex comparison criteria, the only constraint being that the criterion must be expressible by means of a query.

A final remark concerns the process to be used for managing models and measures. Since the database will generally contain data from several model versions, it is advisable that a configuration management system is used to track the association of models with related data. In particular, the user must make sure that for every model version the uuid is univocally defined.

6 A MEASUREMENT PROCESS SUPPORTED BY UMMT

We tested UMMT by measuring several models developed both by means of Rational Rose [Rose] and Argo/UML [Argo/UML]. In particular, we measured the model of a library management program that we use as an example when teaching UML. The model is not reported here for space reasons.

Fig.5 shows the screen of the MySQL Control Center being used to write and execute measurement queries on the database containing the relevant information extracted from the model by means of UMMT.

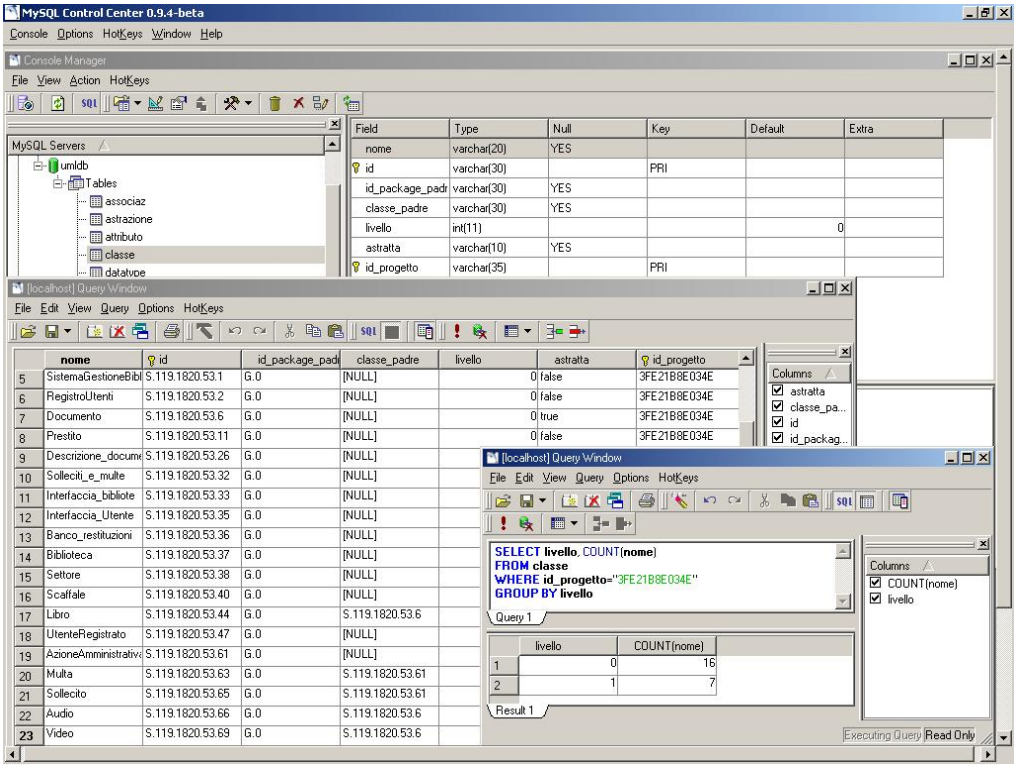


Fig. 5. Measurement of the model by means of MySQL Control Center.

In particular, in the background of it is possible to see the list of tables belonging to the umldb database: table "classe" [1] is selected and its definition appears in the right hand side of the window. Two query windows are also visible: the window on the left reports the list of the classes belonging to the model, while the window on the right shows both the definition and the result of the query. The definition of the shown query is

```
SELECT livello, COUNT(nome)
FROM classe
WHERE id_progetto="3FE21B8E034E"
GROUP BY livello
```

It counts the classes at the same level ('livello' in Italian) of the generalization hierarchy. It is possible to see that in the examined model we have 16 root classes and 7 classes that are children of root classes. No classes are at levels greater than 1.

In order to assess the suitability of UMMT to support real and demanding development processes, we employed the tool in a project carried out in the Mobile Communication department of a big telecom company. The object of the measurement activity were three versions of the models of a GSM network management system. The size of the models ranged from 4312 to 5029 classes [Denaro03].

[1] Currently the schema of the database employed in the UMMT is written in Italian. The terms "classe", "livello", "nome", "id_progetto" mean, respectively, "class", "level", "name" and "id_project". The translation of the schema into English is in progress.

In this case, the process owners were not interested only in the measures of the models; they also wanted to explore the relationships between the measures concerning the models and the measures of the corresponding C++ code. It must be noted that while the UML models could be adequately characterized by a suitable subset of the Chidamber&Kemerer metrics already supported by the UMMT, such metrics were not sufficient to fully characterize the C++ code. In fact, the process owners indicated explicitly that they were interested in the widest possible set of metrics, including both the Chidamber&Kemerer metrics and more traditional ones, namely LOCs and cyclomatic complexity.

In order to extract the required measures, the measurement process was organized as illustrated in Fig. 6. The C++ code was measured by means of a commercial tool. In particular, every C++ versioned file was measured. In order to perform analyses involving both the measures of the models and the measures of the code, we loaded the latter into the UMMT database. In this case, the openness of the UMMT played an important role, as we were able to load the UMMT database with the data provided by the C++ measurement tool with minimum effort. Also writing queries to combine the model and code measures was relatively straightforward.

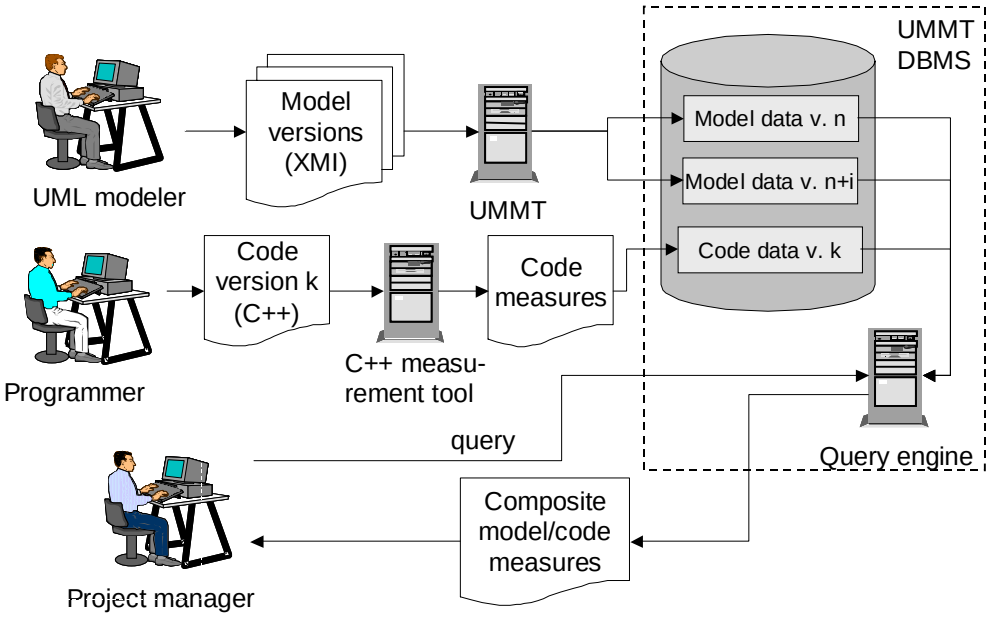


Fig. 6. Comparison of model and code measures.

The UMMT worked quite satisfactorily, and its outcome was useful to study both the relations existing between different versions of the models and between models and the corresponding code. As an example, Fig. 7 reports the comparison of the number of methods per class in two model versions, considering only classes present in both versions. These data were obtained easily from the UMMT.

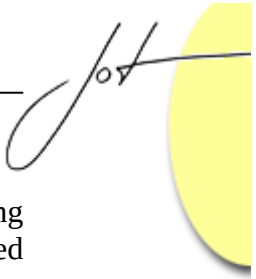


Fig. 8 reports the comparison of the number of methods per class, considering classes present in both the model and the corresponding code. These data can be obtained combining the measures performed by the UMMT with the measures obtained from an external C++ measurement tool.

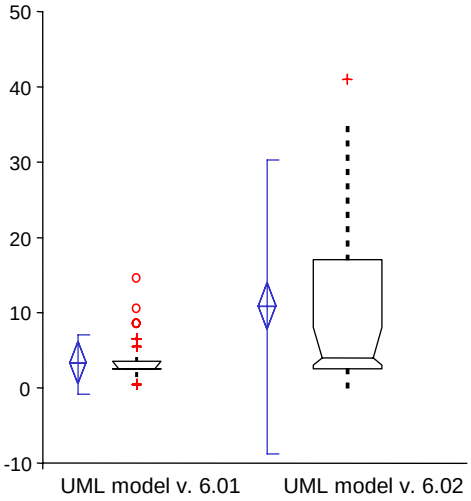


Fig. 7. Number of methods per class: comparison of versions 6.01 and 6.02.

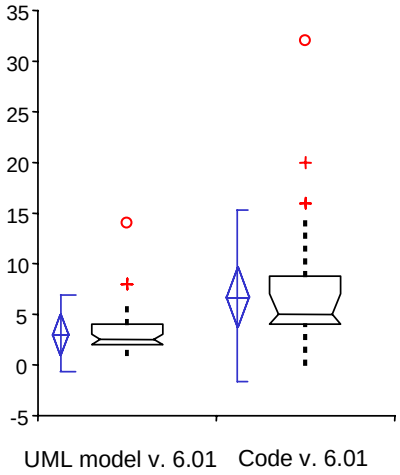


Fig. 8. Number of methods per class: comparison of model and code measures, version 6.01.

The experience briefly sketched above was useful to confirm the effectiveness of UMMT (the tool was able to measure models including thousands of classes in a very reasonable time). Moreover, we were able to appreciate the flexibility provided by the openness of the measures database. In fact, we were able to load the UMMT database with all the needed data –which involved extending the schema of the database suitably– and, more important, we could write ad-hoc queries to perform the computations involving the C++ measures.

7 RELATED WORK

Several tools for the measurement of UML models are able to analyze only the models produced by a specific UML editor. For instance:

- Fast&Serious [Carbone02] automatically extracts data about the project under analysis from Rational Rose. After data about a project has been extracted from UML diagrams, a measurement process is applied that computes for each class a size estimation, which in its turn is used to estimate the development effort.
- UMP (UML Metrics Producer) [Kim02] was developed on top of Rational Rose using its BasicScript language. It computes a set of software metrics (partly defined ad-hoc) that are used to predict various characteristics of the software product in the early stages of the life cycle.

Both of these tools can be applied only to Rose models; they do not allow the user to specify new metrics (unless by re-programming the tools) and do not take into consideration the measurement of variations.

Objecteering/Metrics is a module of the Objecteering UML modeling tool that can be used to derive metrics from a UML model built with Objecteering [Objecteering]. The metrics collected are the usual counting metrics (e.g., number of classes, methods, etc.) and a set of quality metrics (mainly similar to those proposed by Chidamber and Kemerer). The tool does not allow the user to define new metrics. Similarly, it does not support the comparison of different model versions.

Metrics from XMI [Paterson02] is a Java applications that derives object-oriented metrics from the XMI representations of UML models. However, the tool has severe limitations with respect to the requirements stated in Section 1: it does not support the measurement of variations, it does not support measurement of state diagrams, it does not allow the user to define new metrics, and it does not export the results of measures in easy-to-use formats.

SDMetrics [SDMetrics] is a quite mature tool that supports the measurement of several UML diagrams, including activity diagrams and use cases. It even supports the measurement of differences between two versions of UML models.

SDMetrics shares several features with UMMT, for instance it also derives object-oriented metrics from the XMI representations of UML models. The main drawback of SDMetrics with respect to UMMT is that it is not as open. In particular, the measures repository is not directly available to the user. This limits the usability of the tool in several respect. For instance, with SDMetrics it is only possible to compare two versions of a model. Computing the average and the standard deviation of the increment in classes among several successive versions of a model is not possible. Similarly, analyzing measures of models and code together is not possible. For this purpose, the suggestion from the developer of SDMetrics is to reverse-engineer the code into a UML model, and then measure this model. In this way, however, some of the most interesting characteristics of the code (e.g., LOCs, cyclomatic complexity, etc.) are lost.

Actually SDMetrics allows the user to export measures in several formats, so that it is possible to load an external database with model data from SDMetrics and code data from some other tool. However, in this way the “real work” is carried out of SDMetrics.

In conclusion, we believe that UMMT is a step forward with respect to the existing tools. SDMetrics is probably a better choice than UMMT for routine work. However, when specific needs arise, requiring to handle more data than SDMetrics can manage, as in the cases described in Section 6, the openness of UMMT –e.g., the possibility to extend the schema of the database, to write new queries, and even to write programs that interface with the UMMT database–is a clear advantage. For instance, since the relational calculus is not computationally complete, it is clearly possible to devise metrics that cannot be expressed by means of queries. In order to perform this kind of metrics a program interfaced with the database is needed: with our approach writing such program is possible and relatively easy.

8 CONCLUSIONS AND FUTURE WORK

In the introduction we reported the requirements for a UML measurement tool:

- Independence form the tool used to build the models.
- Support of user-defined metrics.
- Measure the differences between versions of the same model.
- The combination of UML metrics with other kinds of metrics should be possible, in order to support the measurement of the artifacts produced in the various phases of the development process.
- Interoperability: the outcome of the measurement tool should be usable by other tools, e.g., tools performing statistical analysis.

UMMT, the tool described in the paper, satisfies all of the requirements described above. UMMT is built according to a flexible, open approach, based on two steps: first the relevant data concerning a model are retrieved from the model itself and loaded into a database, then measures are derived by querying the database. This approach allows the user to load data concerning different versions of the model, and even data concerning the code, and then measure differences, similarities, and whatever property is considered of interest. UMMT is going to be released under the GPL license. Currently we are preparing the distribution package and writing some the documentation.

Future work

In Section 6 we have shown that the proposed approach can be applied to compare a model with the corresponding implementation code. However, in Section 6 we adopted an ad-hoc approach, employing an external tool for code measurement, and importing the results in the UMMT database. This process required –beside acquiring a measurement tool– to extend the schema of the database and to write some additional queries.

Since most of the measures of the code have the same meaning and definition of those derived from UML models, we can aim at a tighter integration of code measurement in the UMMT. In particular, it is possible to conceive a common meta-model that applies both to object-oriented models and code: therefore it would also be possible to write the corresponding database schema that makes it possible to manage in an integrated way both code and model measures. This integrated approach is illustrated in Fig. 9 (in the figure only Java is mentioned, but the approach can be applied to object-oriented code in general). Note that since there is a unique meta-model for code and models, a unique version of UMMT is able to handle both code and models. This new UMMT does not need to be dramatically different from the current one, e.g., object-oriented code can be represented by means of XML, much like UML models. Tools that convert Java code into XML are already available [BeautyJ], therefore we could exploit them to get a representation in XML of the Java code. Then we should just enhance

UMMT to analyse XML files and load the database with the required information. The same queries that are applied to the model data could be applied to the code data, implementing the same measures. The realization of such a tool is among our future objectives.

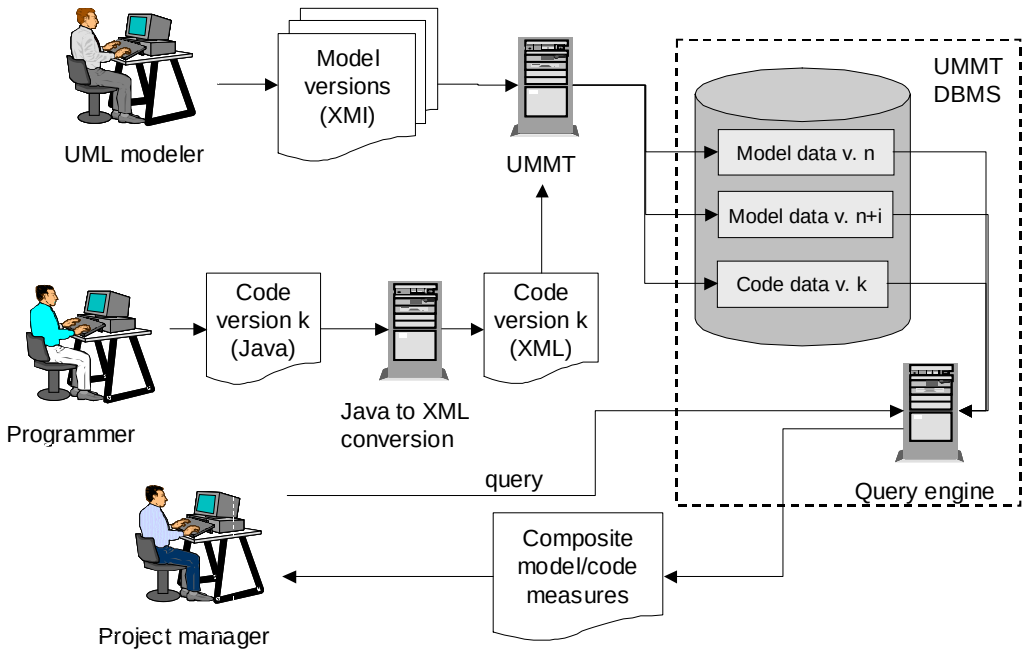


Fig. 9. UMMT for model and code measurement.

Finally, if the usage of UMMT will indicate the necessity of analyzing additional elements of UML models (e.g., use cases), we shall enhance the tool in order to read from XMI files the needed data, and load them in the measurement database. In particular we shall consider the possibility of defining and supporting measures covering OCL: this could be useful to support measures concerning requirements and design for testing, where OCL is most often employed.

REFERENCES

[OMG03] *OMG Unified Modeling Language Specification Version 1.5*, March 2003, formal/03-03-01. http://www.omg.org.

[OMG00] *OMG XML Metadata Interchange Specification Version 1.1*, November 2000.

[Agostini03] Alberto Agostini: *Uno strumento per la misurazione di modelli UML*, Tesina di Laurea, Politecnico di Milano, July 2003 (in Italian).

[Chidamber94] S.R. Chidamber and C.F. Kemerer: "A Metric Suite for Object-Oriented Design", *IEEE Trans. Software Eng.*, vol.20, no.6, pp.476-493, June1994.

[Cartwright00] M. Cartwright and M. Shepperd, "An Empirical Investigation of an Object-Oriented Software System", *IEEE Trans. Software Eng.*, vol.26, no.8, pp.786-796, August 2000.

[Argo/UML] http://argouml.tigris.org/

[Rose] http://www.rational.com/products/rose/index.jsp

[Carbone02] M. Carbone and G. Santucci: "Fast & Serious: a UML Based Metric for Effort Estimation", *6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, June 11th, 2002.

[Kim02] H. Kim, C. Boldyreff: "Developing Software Metrics Applicable to UML Models" , *6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, June 11th, 2002.

[SDMetrics] http://www.sdmetrics.com

[Paterson02] T. Paterson: *Object-Oriented Software Design Metrics from XMI* MSc Dissertation, Heriot-Watt University, 2002.

[Objecteering] http://www.objecteering.com

[Lorenz94] M. Lorenz and J. Kidd: *Object-oriented Software Metrics,* Prentice-Hall Object Oriented Series, 1994.

[MySQL] http://www.mysql.com/

[Xerces] http://xml.apache.org/#xerces

[Purao03] S. Purao and V. Vaishnavi: "Product Metrics for Object-Oriented Systems", *ACM Computing Surveys*, vol. 35, no. 2, pp. 191–221, June 2003.

[Basili96] V.R. Basili, L.C. Briand and W.L. Melo: "A validation of object-oriented design metrics as quality indicators", *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp.751-761, 1996.

[Chidamber98] Chidamber, S.R., Darcy, D.P. and Kemerer, C.F. "Managerial use of metrics for object-oriented software: and exploratory analysis", *IEEE Transactions on Software Engineering*, 24, n. 8, 1998, pp.629-639.

[BeautyJ] http://beautyj.berlios.de/

[Denaro03] G. Denaro, L. Lavazza and M. Pezzè, "An Empirical Evaluation of Object Oriented Metrics in Industrial Setting", *The 5th CaberNet Plenary Workshop*, Porto Santo, Madeira Archipelago, Portugal, November 2003.

About the authors

Luigi Lavazza received his Dr. Eng. degree in Electronic Engineering from Politecnico di Milano in 1984. He is currently an assistant professor at Politecnico di Milano, Dipartimento di Elettronica e Informazione, where he carries out his research activity in the Software Engineering group. Since 1990 he is a member of the Software Engineering research group at CEFRIEL. Here he leads research projects as well as consultancy activities and technology transfer initiatives.

His research interests include advanced software engineering environments, software process modeling, assessment, improvement and measurement, and requirements engineering. Additional information are available from http://www.cefriel.it/~lavazza, E-Mail: lavazza@cefriel.it

Alberto Agostini received his Dr. Eng. degree in Electronic Engineering from Politecnico di Milano in 2003. The work described here is the subject of his Thesis. He is currently involved in the development of application software supporting the call center of a big telecommunication company. E-mail: albago@inwind.it