DEQUALITE: Building Design-based Software Quality Models

Foutse Khomh and Yann-Gaël Guéhéneuc
Ptidej Team, GEODES, DIRO, University of Montreal,
C.P. 6128 succursale Centre Ville Montréal, Quebec, H3C 3J7, Canada

E-mail: {foutsekh,guehene}@iro.umontreal.ca

Abstract

Object-oriented software quality models usually use metrics of classes or of relationships between classes to measure internal attributes of systems. However, the quality of these systems does not depend on classes solely: It also depends on the organisation of classes, i.e.*, their design. We propose DEQUALITE, a method to build models to measure the quality of systems taking into account both their internal attributes and their designs. Our method uses a machine learning approach and also allows combining different models to improve the quality prediction. In this paper, we justify the use of patterns to build quality models, we illustrate our method on a set of systems implementing design patterns and on the quality model QMOOD from Bansiya* et al. *We discuss the advantages and limitations of this method, we then present a validation of a resulting quality model on a set of systems. We conclude on the advantages of using patterns to build models and the difficulty of doing so.*

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*

General Terms

Design, Measurement

Keywords

Patterns, Quality, Machine learning, Models

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission. A preliminary version of this paper was presented in a writers' workshop at the 15th Conference on Pattern Languages of Programs (PLoP). *PLoP '08,* October 18–20, 2008, Nashville, TN, USA.
Copyright 2008 is held by the author(s). ACM 978-1-60558-151-4.

1. Introduction

In this paper, we present a method, DEQUALITE (Design Enhanced QUALITy Evaluation), to build quality models to measure the quality of object-oriented systems by taking into account both their internal attributes and their designs.

Maintenance cost of systems during the past decades increased to more than 70 % of the overall cost of systems [21] because of many factors such as changes in software environment, new users requirements, and quality of systems [2]. One factor on which we have a real control is the quality of systems; thus, being able to assess the quality of systems can significantly help in the prediction of maintenance effort.

Quality models link software artifacts with quality characteristics to predict system quality. Many quality models exist in the literature but none of them takes into account the systems designs during their evaluations. They focus on the internal attributes of classes (such as size, filiation and cohesion) or, at best, of pairs thereof, and disregard their organization. They thus can hardly distinguish between a well-structured system and a system with poor design even though their designs are the first things that software engineers see.

There are many principles and techniques to help design systems with good quality characteristics; among these, design patterns are an interesting bridge between internal attributes of systems, external quality characteristics, and software designs because they link internal attributes (concrete implementation of systems) and subjective quality characteristics (subjective perceptions on systems), such as reusability [11].

Since their popularisation in the software engineering community in 1994, design patterns have gained importance for system design and have been the subject of many studies on quality. Some authors like Venners claimed that design patterns improve the quality of systems while others like Wendorff [24], suggested that their use do not always result in "good" designs. MacNatt *et al.* [20] showed that a tangled implementation of patterns impacts negatively quality.

In this paper, we present DEQUALITE, a method to build quality models to measure the quality of object-oriented systems taking into account both their internal attributes

and their designs, see Figure 1. This method stands on our previous study of the impact of design patterns on quality [22] and on the use of a machine learning technique to link internal attributes and subjective quality characteristics. We also use a combination technique [4] to build composite models with greater predictive power.

In addition to this method, we introduce two other contributions:

- we present a quality model PQMOD. This quality model is composed of a set of rules for the evaluation of quality taking in account design. We discuss these rules with respect to object-oriented principles and present the results of their application on a set of systems including design patterns;
- we reuse a technique to combine PQMOD with a rule-based quality model from the literature to improve the predictive power of both models with the expertise of design patterns contained in PQMOD. We illustrate this combination on the model QMOOD [1].

Our method uses the Dromey's approach [9] that involves four steps to build a quality model:

1. Identify a set of high-level quality attributes.
2. Identify and classify the most significant, tangible, quality-carrying properties of the system.
3. Propose a set of axioms for linking product properties to quality attributes.
4. Evaluate the model, identify its weaknesses, and either refine it or scrap it and start again.

We organized this paper along the steps of the Figure 1. Section 2 presents the first part of the method DEQUALITE that is an empirical study of the impact of the Gamma *et al.*'s 23 design patterns [11] on ten quality attributes of systems. Section 3 presents PQMOD, an exemple of quality model that takes in account design during its evaluation. Section 4 presents a technique to improve quality models. Section 5 presents related work. Section 6 concludes our work.

2. Part 1: Why Design Patterns?

Using design patterns can be viewed as programming style from an artistic point of view. In fact, if we draw a parallel with art, and view a system as a painting, design patterns can be viewed as the style of the painting and the quality of this style has an impact on the quality of the painting, regardless of what is shown. For exemple, if we want to compare a cubist painting, such as "Femme Profile" by Pablo Picasso (1939), with a realistic picture (as shown in figure 2), we should consider the style of each picture because the two faces possess two eyes, one nose, two ears, and one mouth only with very different organisations.

We think that similarly to art, system quality depends on its design; that is on the organisation of its constituents, which is often determined by design patterns. The link between paintings and systems is shown in Table 1: Painting relates to a system; eyes, nose, ears, mouth to classes, interfaces, and methods; the organisation of the painting to the design of the system; the style of the painting to the patterns; art critics to quality engineers; and, painters to developers.

Figure 2. A woman's profile: Realist version and Cubist

Art	Development
Painting	System
Organisation	Architecture
Eyes, nose, ears, mouth	Classes, interfaces, methods, etc.
Style	Patterns
Art critics	Quality engineers
Painters	Developers

Table 1. Parallel between painting and systems design.

The study of the impact of design patterns on system quality is necessary to take into account patterns in the assessment of systems and is similar to the study of the impact of an artist's style on her paintings.

We have evaluated the impact of Gamma *et al.*'s 23 patterns [11] on ten quality attributes and concluded that they have a quantifiable impact on system quality. We recall here some results of this study [22].

2.1. Quality Attributes

We chose [22] the following quality attributes, based on their relevance to patterns.

- Attributes related to design:
  - **Expandability:** The degree to which the design of a system can be extended.
  - **Simplicity:** The degree to which the design of a system can be understood easily.
  - **Reusability:** The degree to which a piece of design can be reused in another design.
- Attributes related to implementation:
  - **Learnability:** The degree to which the code source of a system is easy to learn.
  - **Understandability:** The degree to which the code source can be understood easily.

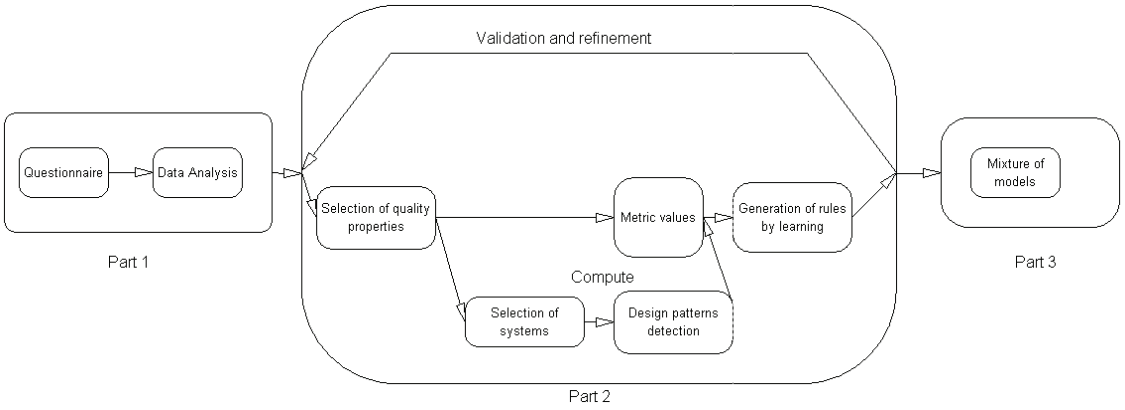

Figure 1. Steps of the method DEQUALITE.

  - **Modularity:** The degree to which the implementation of the functions of a system are independent from one another.

- Attributes related to runtime:
  - **Generality:** The degree to which a system provides a wide range of functions at runtime.
  - **Modularity at runtime:** The degree to which the functions of a system are independent from one another at runtime.
  - **Scalability:** The degree to which the system can cope with large amount of data and computation at runtime.
  - **Robustness:** The degree to which a system continues to function properly under abnormal conditions or circumstances.

Each quality attribute was evaluated using a six-point Likert scale: `A - Very positive`, `B - Positive`, `C - Not significant`, `D - Negative`, `E - Very Negative`, and `F - Not applicable`. The sixth value allowed respondents not to answer a question if they did not know or were not sure about the impact of a design pattern on a quality attribute.

For every design pattern in [11] and for every quality attribute from our set, the respondents were asked to assess the impact of the pattern on the quality of a system in which the pattern would be used appropriately, as they would during a technical review [21] or possibly while performing a program comprehension-related activity during maintenance and evolution.

The questionnaire is available on the Internet at `http://www.ptidej.net/downloads/`.

2.2. Data Collection and Processing

We collected respondents' evaluations during the period of January to April 2007 by posting our questionnaire on three mailing lists, *refactoring*, *patterns-discussion*, and *gang-of-4-patterns*.

Among the many answers that we received, we selected the questionnaires of 20 software engineers with a verifiable experience in the use of design patterns in software development and maintenance.

This number of collected evaluations is larger than in previous work. Due to the variations between answers, we felt that the differences between `Positive` and `Very Positive` answers were due to some respondents being less strict than others and thus, that their `Very Positive` evaluations were not directly relevant. This fact has been confirmed in discussions with the respondents. For example, for Builder and expandability, we had 19% of respondents considering the pattern `Very Positive` while 63% considered it `Positive` and 18% considered it `Neutral`. Therefore, we chose to aggregate answers `A` and `B` and answers `D` and `E`: `G = A` and `B`, `N = C`, and `B = D` and `E`. Where `G` stands for Good, `N` for Neutral and `B` for Bad.

Using the previous three-point Likert scale, we computed the frequencies of the answers on each quality attribute: `G`, `N`, and `B` and we carried out a Null hypothesis test to assess the perceived impact of the patterns on the quality attributes.

Answers `F` were not considered because they represented situations where the respondents did not know or did not want to evaluate the impact.

2.3. Results

Using the results collected from the questionnaires and presented in details elsewhere [17], we carried out Null hypothesis tests to quantify the impact of the design patterns on the quality attributes and then confirm or refute the hypothesis that *design patterns impact software quality positively.*

The null hypothesis test yields some surprising results as one can see in [22], the following table summarize the results for all the 23 design patterns from [11].

3. Part 2: A Quality Model that takes into account Design Patterns

This second part of the method DEQUALITE involves five steps among which three consist of preliminaries to the construction of the model. The other two are the construction and the validation of the model. First, we have chosen a set of appropriate quality characteristics in the previous part. Second, we identify and classify the most significant, tangible, internal attributes of systems implementing design patterns. Third, we assess these attributes on sys-

Design patterns	Quality attributes									
	Expandability	Simplicity	Generality	Modularity	Modularity at runtime	Learnability	Understandability	Reusability	Scalability	Robustness
A.Factory	G	G	G	G	G	B	G	G	N	N
Builder	G	G	G	G	G	G	G	N	N	N
F.Method	G	G	G	G	G	G	G	G	N	N
Prototype	G	G	G	N	G	G	G	G	N	N
Singleton	B	G	N	N	N	G	G	N	N	N
Adapter	G	G	G	G	G	G	G	G	B	N
Bridge	G	B	G	G	G	B	G	B	B	N
Composite	G	G	G	G	G	G	G	G	G	N
Decorator	G	N	G	G	G	G	B	B	B	N
Facade	G	G	G	G	B	G	G	N	N	N
Flyweight	N	B	B	B	N	B	B	B	G	N
Proxy	N	G	G	G	N	G	N	G	B	N
Ch.Of.Resp	G	G	G	G	G	G	B	G	N	N
Command	G	G	G	G	G	G	B	B	B	N
Interpreter	G	G	G	G	G	G	G	G	B	N
Iterator	G	G	G	G	G	G	G	G	B	N
Mediator	G	B	G	G	N	G	G	N	N	N
Memento	N	G	N	N	N	G	N	G	B	N
Observer	G	G	G	G	G	G	G	G	N	N
State	G	G	G	G	G	G	G	B	N	N
Strategy	G	G	G	G	G	G	G	G	N	N
T.Method	G	G	G	N	N	G	G	G	N	N
Visitor	G	B	G	G	G	B	B	B	B	N

Table 2. Evaluation of the impact of design patterns on the quality of systems (G = Good, N = Neutral, B = Bad).

tems. Fourth, with the evaluation of the impact of design patterns on the quality presented in Table 2, we apply machine learning techniques (JRip and J48) to generate a set of rules that link internal attributes to quality characteristics while taking into account the impact of design patterns. Finally, fifth, we carry a validation and a refinement of the resulting quality model.

3.1 Steps 1, 2, and 3: Preliminaries

Selection of Internal Attributes of Systems. This step consist of selecting internal attributes of systems that relate to the quality characteristics in Subsection 2.1 and that can be measured. We choose 29 metrics from the literature [5, 8, 15, 18, 23], among which metrics of coupling, cohesion, size, filiation, complexity, number of methods... The metrics are implemented in the POM framework [13].

Selection of Systems Implementing Design Patterns. We select a set of systems implementing design patterns. We call this set $\mathcal{BS}$. It includes: QuickUML 2001, Lexi v0.1.1, JRefactory v2.6.24, Netbeans v1.0, JUnit v3.7, JHotDraw v5.1, MapperXML v1.9.7, Nutch v0.4, PMD v1.8. Table 3 presents a detailed list of the design patterns contained in these systems and their numbers. We carried a manual identification of design patterns in the systems, which results have been stored in an XML database [3, 13], to obtain a good precision and recall. An automatic detection would have produced many false positives and duplications of design patterns occurrences.

Measurement of the Internal Attributes of $\mathcal{BS}$. We compute the selected metrics on the classes playing roles in the patterns in the systems from $\mathcal{BS}$.

3.2 Step 4: Construction of Rules

We apply machine learning techniques, JRip and J48 [25], to infer rules linking the internal attributes with the metric values obtained in Step 3. We select the rules with higher classification rates. We obtain a quality model named PQMOD. The next paragraph presents an example of a rule in PQMOD:

Rule for the Expendability.

```
NOA <= 2
|  AID <= 0.5: Value: N
|  AID > 0.5: Value: B
NOA > 2
| NOC <= 0
|  |  DIT <= 0.83: Value: G
|  |  DIT > 0.83: Value: B
| NOC > 0: Value: G
```

The classification rate for this rule is 89.36%. This rule states that for a good expendability, the average number of ancestor (NOA) should be > 2 and each class should have at least on descendent. This rule confirms the deep abstract hierarchy principle that is a principle of good programming [19].

3.3 Step 5: Validation of PQMOD

At this step, we performed a double validation. First, we applied the rules of PQMOD on design patterns implemented in the system PADL [12] and compared the results with those of the table 2. We obtained the expected results for the patterns Visitor, Observer, A.Factory and Composite contained in PADL. Which confirm the accuracy of our rules at a pattern level. Next, we scaled the rules of PQMOD (the next section present the details of the scaling) to be able to apply them to a system at whole. We apply the scaled model on 5 systems and compared the results to a manual evaluation of the systems by a group of independent experts.

Scaling of PQMOD. We scale the quality model PQMOD to apply it on whole systems. This step consists in adapting the rules associated with PQMOD. Indeed, the rules are built from metric values with a certain minimum and maximum values depending on $\mathcal{BS}$, these values differ from the minimum and maximum values of a given system $\mathcal{S}$. We compute the ratio between $min_{\mathcal{BS}}$ and $max_{\mathcal{BS}}$, on the one hand, and $min_{\mathcal{S}}$ and $max_{\mathcal{S}}$, on the other hand, to adapt their scales. Figure 3 illustrates this adaptation. The metrics values of $\mathcal{S}$ are thus on the same scale with those of $\mathcal{BS}$ and we can apply PQMOD on any system $\mathcal{S}$.

Systems	List of design patterns	Number of occurrences
QuickUML 2001	A.Factory, Builder, Command, Composite, Observer, Singleton	7
Lexi v0.1.1	Builder, Observer, Singleton	5
JRefactory v2.6.24	Adapter, Builder, F.Method, Singleton, State, Visitor	26
Netbeans v1.0	A.Factory, Adapter, Command, Iterator, Observer	28
JUnitv3.7	Composite, Decorator, Iterator, Observer, Singleton	8
JHotDraw v5.1	Adapter, Command, composite, Decorator, F.Method, Observer, Prototype, Singleton, State, Strategy, T.Method	24
MapperXML v1.9.7	A.Factory, Adapter, Composite, Facade, F.Method, Observer, Singleton, Strategy, T.Method	16
Nutch v0.4	Singleton, Bridge, Command, Memento, T.Method, Adapter, Strategy, Iterator	16
PMD v1.8	Adapter, Bridge, Composite, F.Method	8

Table 3. Systems from our $\mathcal{BS}$.

Quality attributes	Nutch v0.4		Xerces v 1.4.4		Ant v1.7.0		GanttP. v2.0.4		PADL	
	Expected	Predicted	Expected	Predicted	Expected	Predicted	Expected	Predicted	Expected	Predicted
Expandability	G	G	G	G	G	G	B	G	G	G
Generality	N	G	G	G	G	G	B	G	N	G
Modularity	G	G	G	G	G	G	N	G	G	G
Modularity at runtime	G	G	N	G	G	N	B	N	N	G
Understandability	N	G	N	G	G	N	N	G	N	G
Reusability	G	G	N	G	G	G	B	G	N	G
Scalability	G	B	G	N	G	N	G	B	B	B

Table 4. Results of the application of PQMOD on 5 systems.

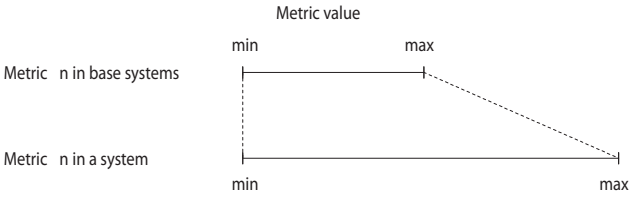

Figure 3. Adapting the rules of PQMOD, ratio between minimum and maximum metric values of $\mathcal{BS}$ and $\mathcal{S}$.

Analysis of PQMOD Results. Table 4 presents the results of applying PQMOD on five open source systems. We observe that PQMOD performs better on systems with a sizable number of design patterns. Consequently, we conclude on the need to complement PQMOD with the expertise of another quality model from the literature to improve their respective predictive powers. We choose a quality model that performs well on systems without design patterns. The quality model PQMOD assesses the quality of systems by comparing their structure to structures of design patterns.

4. Part 3: Mixture of Quality Models

The technique of mixture of expertise allows every rule-based quality models from the literature to be improved by the expertise of PQMOD and vice-versa. We now present the third part of our method that reuse a technique of mixture of experts to produce quality model that takes into account the design of systems through design patterns.

Notations. Let T be a given decision tree, we call $\mathcal{V}$ the domain of the function $f : \mathcal{V} \mapsto \mathcal{C}$ that predicts the value $y_i \in \mathcal{C}$ of a quality factor y for an observation $x_i \in \mathcal{V}$. $\mathcal{V}$ is defined by the cartesian product $A_1 \times \ldots \times A_d$, where A_j is the domain of the j^{th} attribute and d is the number of attributes. The range $\mathcal{C}$ is generally and ordered set $c_1, c_2, \ldots, c_q$ of labels.

Each internal node of T is a test of the type $x^{(j)} < \alpha_j$, with $j = 1 \ldots d$, $x^{(j)}$ an attribute of the observation x and α_j a constant belonging to the domain of the attribute $x^{(j)}$. Each attribute $x^{(j)}$ takes its values into a bounded domain with L_j and U_j being, respectively, the lower and upper

bounds.

The tree T can be represented by a binary partition S of a space $\mathcal{V} \subset \mathrm{R}^d$ defined by the cartesian product of the intervals $[L_j, U_j], j = 1 \ldots d : \mathcal{V} = [L_1, U_1] \times \ldots \times [L_d, U_d]$.

The partitioning S is equivalent to subdividing recursively in two sub-spaces, the space $\mathcal{V}$ with an hyperplane h_j of equation $x^{(j)} = \alpha_j$. An hyperplane h_j is associated to each internal test node $x^{(j)} < \alpha_j$. The subdividing continues until every sub-space is composed only of points belonging to the same class. The partitioning S enables us to view a decision tree as a set of "isothetic" hyperrectangles (that is with faces parallel to the axes of the attributes). Thus, a decision tree can be represented by a vector of (hyperrectangle, label). An hyperrectangle (also call a d-rectangle) R_k is defined by the cartesian product of the intervals $[l_j, u_j]$, $R_k = [l_1, u_1] \times \ldots \times [l_d, u_d]$, with l_j and u_j being the coordinates of the edges of the d-rectangle, on the axis of the attribute $x^{(j)}$.

Approach. We adopt a process of model combination and adaptation based on the three following steps:

1. Decompose the models to be combined into expertise chunks to ensure the interpretability of the resulting model.
2. Combine expertises chunks to improve the generalisability of the resulting model.
3. Adapt/calibrate expertise chunks to improve the predictive power of the resulting model.

To decompose each model into a set of expertise chunks, we consider, as a criterion of decomposition, the variation of the predictive power of a model from an area to another in its input space. To illustrate our mixture of expertise, let us consider two decision trees predicting the quality characteristics Stability, for exemple. The first model uses two metrics, NPPM and LCOMB, and the second one uses the metrics DIT and LCOMB. The decomposition of the first decision tree into expertise chunks yields the result in Figure 4.

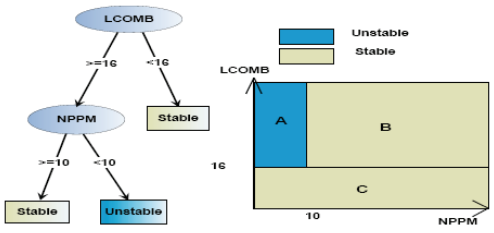

Figure 4. A two-dimensional example of decision-tree chunk of expertise(this figure is from (Bouktif, 2005))

The combinaison of the expertises chunks of the two decision trees is illustrated in Figure 5.

After this combination of expertises chunks, a unique decision tree is obtained by combining the sets of d-rectangles

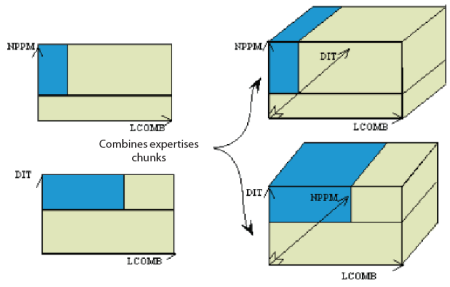

Figure 5. Combination of chunks of the two decision trees (from [4])

of all the decision trees. To achieve this, we compare the predictive power of each d-rectangles for the two decisions trees and select the d-rectangle with highest power. The details of this technique can be found in [4].

Application. Applying this technique of mixture of expertise on the quality models PQMOD presented in Part 2 and QMOOD from Bansiya [1], we obtain the following result for the rule Expendability:

```
NOA <= 2
|  AID <= 0.5 : Value: N
|  AID > 0.5 : Value: B
NOA > 2
| Value: (0.5)* ANA-(0.5)* DCC+
  (0.5)* MFA+(0.5)* NOPM
```

This rule combines the predictive power of the model PQMOD (that takes into account the design of systems) and the predictive power of the model QMOOD (that has been validated in the literature on many systems but that does not take into account directly the design of systems). Thus, by combining the quality model PQMOD and rule-based models from the literature, the method DEQUALITE enables the construction of quality models that take into account the design of systems and that perform at least as good as the models from the literature.

5. Related Work

We present some major work on quality models and show that none of the existing work attempts to build a quality model while considering design.

Briand and Wüst [6] present a detailed and extensive survey of quality models. They classify quality models in two categories: correlational studies and experiments. Correlational studies use univariate and multivariate analyses, while experiments use, for examples, analysis of variance between groups (ANOVA). To the best of our knowledge, none of the presented quality models attempts to assess the architectural quality of programs directly. They all use class-based metrics or metrics on pairs of classes.

Harrison et al. [14] investigate the structure of object-oriented programs to relate modifiability and understandability with levels of inheritance. Modifiability and under-

standability cover only partially quality characteristics related to maintenance. Levels of inheritance are but one architectural characteristic of programs related to software maintenance.

Wydaeghe et al. [26] assess the quality characteristics of the architecture of an OMT editor through the study of 7 design patterns. They conclude on flexibility, modularity, reusability, and understandability of the architecture and the patterns. However, they do not link their assessment with any quality model.

Although some studies have assess some architectural characteristics of program none have attempted to build a predictive quality model.

6. Conclusion

In this paper, we have proposed DEQUALITE, a method to build quality models that allows the measurement of the quality of object-oriented systems by taking into account the internal attributes of the system and also its design. Our method uses a machine learning approach and enables the combination of models for the improvement of the performance of resulting quality models. Our method is divided in three parts and 8 steps. We have validated the different steps of our method on many systems. The use of design patterns in building a quality model brings an extra level of abstraction to the resulting quality models of the method DEQUALITE.

The use of design patterns is an important step toward the construction of quality models able to assess the quality of a system by taking into account not only its internal attributes and its design, but also more detailed architectural informations like the density of patterns and–or the presence of anti-patterns [7] and code smells [10] for exemple.

In our future work, we plan to improve our method with architectural metrics as suggested by [16]. We are thinking on using density of patterns, composition of patterns, anti-patterns and code smells... We also plan to validate our resulting quality models on a larger number of systems.

Acknowledgments

This work has been partially funded by NSERC and the VINCI program of University of Montreal. “Femme Profile” by Pablo Picasso is from rogallery.com, we are in contact with its director regarding copyrights.

7. References

[1] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. In *IEEE Transactions on Software Engineering*, 28:4–17, January 2002.

[2] K. H. Bennett and V. Rajlich. Software maintenance and evolution: a roadmap. In *The Future of Software Engineering*. ACM Press, 2000.

[3] J. Bieman, G. Straw, H. Wang, P. W. Munger, and R. T. Alexander. Design patterns and change proneness: An examination of five evolving systems. In *Proceedings of the* 9^{th} *international Software Metrics Symposium*, pages 40–49. IEEE Computer Society Press, September 2003.

[4] S. Bouktif. *Amélioration de la prédiction de la qualité du logiciel par combinaison et adaptation de modèles*. PhD thesis, Université de Montréal, Mai 2005.

[5] L. Briand, P. Devanbu, and W. Melo. An investigation into coupling measures for C++. In *Proceedings of the* 19^{th} *International Conference on Software Engineering*, pages 412–421. ACM Press, May 1997.

[6] L. C. Briand and J. Wüst. Empirical studies of quality models in object-oriented systems. In *Advances in Computers*, 59:97–166, June 2002.

[7] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1^{st} edition, March 1998.

[8] S. R. Chidamber and C. F. Kemerer. A metrics suite for object-oriented design. Technical Report E53-315, MIT Sloan School of Management, December 1993.

[9] R. G. Dromey. Cornering the chimera. In *IEEE Software*, 13(1):33–43, January 1996.

[10] M. Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1^{st} edition, June 1999.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1^{st} edition, 1994.

[12] Y.-G. Guéhéneuc. PTIDEJ: Promoting patterns with patterns. In *Proceedings of the* 1^{st} *ECOOP workshop on Building a System using Patterns*. Springer-Verlag, July 2005.

[13] Y.-G. Guéhéneuc, H. Sahraoui, and Farouk Zaidi. Fingerprinting design patterns. In *Proceedings of the* 11^{th} *Working Conference on Reverse Engineering*, pages 172–181. IEEE Computer Society Press, November 2004.

[14] R. Harrison, S. J. Counsell, and R. V. Nithi. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. In *Journal of Systems and Software*, 52(2–3):173–179, June 2000.

[15] M. Hitz and B. Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of the* 3^{rd} *International Symposium on Applied Corporate Computing*, pages 25–27. Texas A & M University, October 1995.

[16] J. Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 1^{st} edition, August 2004.

[17] F. Khomh and Y.-G. Guéhéneuc. An empirical study of design patterns and software quality. Technical Report 1315, University of Montréal, january 2008.

[18] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Approach*. Prentice-Hall, 1^{st} edition, July 1994.

[19] R. C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. 2002.

[20] W. B. McNatt and J. M. Bieman. Coupling of design patterns: Common practices and their benefits. In *Proceedings of the* 25^{th} *Computer Software and Applications Conference*, pages 574–579. IEEE Computer Society Press, October 2001.

[21] R. S. Pressman. *Software Engineering – A Practitioner's Approach*. McGraw-Hill Higher Education, 5^{th} edition, November 2001.

[22] Foutse Khomh and Y.-G. Guéhéneuc. Do design patterns impact software quality positively? In *Proceedings of the* 12^{th} *Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, April 2008. Short Paper.

[23] D. P. Tegarden, S. D. Sheetz, and D. E. Monarchi. A software complexity model of object-oriented systems. In *Decision Support Systems*, 13(3–4):241–262, March 1995.

[24] P. Wendorff. Assessment of design patterns during software reengineering: Lessons learned from a large commercial project. In *Proceedings of* 5^{th} *Conference on Software Maintenance and Reengineering*, pages 77–84. IEEE Computer Society Press, March 2001.

[25] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1^{st} edition, October 1999.

[26] B. Wydaeghe, K. Verschaeve, B. Michiels, B. V. Damme, E. Arckens, and V. Jonckers. Building an OMT-editor using design patterns: An experience report. In *proceedings of the* 26^{th} *Technology of Object-Oriented Languages and Systems conference*, pages 20–32. IEEE Computer Society Press, August 1998. citeseer.ist.psu.edu/wydaeghe98building.html.