21

Quality Model based on Object-oriented Metrics and Naive Bayes

Sai Peck Lee and Chuan Ho Loh
University of Malaya
Malaysia

1. Introduction

Software quality engineering is a field in software engineering specializing on improving the approach to software quality on different software artifacts such as object-oriented analysis models, object-oriented design models, and object-oriented implementation models. Software quality is the degree to which a software artifact exhibits a desired combination of quality-carrying attributes (e.g. testability, reliability, reusability, interoperability, and other quality-carrying attributes). This research specializes on improving the code quality of object-oriented systems through a quality model that utilizes a suite of object-oriented metrics and a machine learning technique, namely Naive Bayes. Most of the existing object-oriented metrics and machine learning techniques capture similar dimensions in the data sets, thus reflecting the fact that many of the object-oriented metrics and machine learning techniques are based on similar hypotheses, properties, and principles. Accurate quality models can be built to predict the quality of object-oriented systems by using a subset of the existing object-oriented design metrics and machine learning techniques. This research proposes a software quality model, namely QUAMO to assess the quality of object-oriented code on-the-fly. The primary objective of the model is to make similar studies on software quality more comparable and repeatable. The model is augmented from five quality models, namely Boehm Model, McCall Model, FURPS, ISO 9126, and Dromey Model. The quality model specializes on Bayesian network classifier, Naive Bayes. The Naive Bayes classifier, a simple classifier based Bayes' law with strong independence assumptions among features is comparable to other state-of-the-art classifiers, namely ID3 Decision Tree, J48 Decision Tree, and C4.5 Decision Tree. Naive Bayes is very effective in solving the classification problems addressed in this research, namely the conditional maximum likelihood prediction of faults in object-oriented systems. Most of the metrics proposed by other researchers mainly specialized at the class level such as CK Metrics Suite and MOOD Metrics Suite (Chidamber & Kemerer, 1994). Fewer component level metrics have been proposed such as Rate of Component Observability, Rate of Component Customizability, and Self-Completeness of Component's Return Value. As such, this research also proposes a suite of specialized object-oriented metrics that can be applied at the class and component levels as some insights can be gained by examining the average characteristics of both a class and a component. Each metric quantifies a particular feature of an object-oriented system. In other words, each refers to a structural mechanism of the

object-oriented paradigm such as inheritance is expressed as quotient. The numerator in the quotient represents the actual use of a mechanism such as inheritance (M_i) on the object-oriented design (OOD). The denominator, which acts as a normalizer, represents the hypothetical maximum use for the mechanism, M_i on an OOD (i.e. it considers the number of classes and their inheritance relations). The metrics are thus expressed as indexes, ranging from 0 (e.g. indicating no use) to 1 (e.g. indicating maximum use).

2. Motivation

A principal objective of software quality engineering is to improve the quality of software artifacts such as object-oriented analysis models, object-oriented design models, and object-oriented implementation models. Quality in software artifacts is a composite of characteristics such as portability, reliability, testability, reusability, and maintainability, which are collectively known as quality-carrying attributes. The factors that affect software quality can be categorized in two distinctive categories, namely factors that can be directly measured (e.g. number of defects) and factors that can be measured only indirectly (e.g. reusability). Generally, the notion of quality is usually captured in the form of a diagram, function or equation, which is collectively known as a model. There are several types of models, namely software process models, software maturity models, and software quality models. Insights to quality can be gained in two ways: by examining quality-carrying attributes through a software quality model, and by examining software artifacts through software metrics (i.e. formulate a set of meaningful software metrics based on these attributes, and to use the metrics as indicators that will lead to a strategy for software quality improvement) and machine learning techniques (i.e. formulate a set of meaningful machine learning techniques based on these attributes, and to use the machine learning techniques as predictors that will lead to a strategy for software quality improvement). In this research, quality is measured in terms of adherence of a set of metrics to a set of attributes used to distinctively evaluate the quality of object-oriented systems by making quality a quantifiable concept via a software quality model.

3. Related work

A number of software quality models have been proposed to evaluate the quality of a software system. The best known software quality models in chronological order are Boehm Model, McCall Model, FURPS, ISO 9126, and Dromey Model (Boehm et al., 1976; McCall et al., 1977; Dromey, 1995, 1996; Ortega et al., 2003). Existing software quality models can be distinguished based on number of layers (e.g. 2 layers as in Dromey model and 3 layers as in Boehm and McCall models), number of relationships (e.g. 1:n relationship as in ISO 9126 model – every characteristic has its own set of subcharacteristics, and n:m relationship as in Factor-Criteria-Model – every subcharacteristic is linked to one or more characteristics), support for metrics (e.g. no support for metrics as in Dromey model and support for metrics as in McCall model), and approach to software quality measurement (e.g. fixed quality model approach as in Boehm, McCall, and ISO 9126 models, and "define your own quality model" approach as in COQUAMO model) (Ortega et al., 2000, 2002, 2003; Callaos & Callaos, 1996; Bansiya & Davis, 2002; Georgiadou, 2003; Khaddaj & Horgan , 2005; Côté et al., 2007). Supervised learning can be formulated using either a discriminative approach (e.g. Logistic Regression) or a generative approach (e.g. Naive Bayes). A number of

supervised learning techniques have been introduced such as Neural Nets, Logistic Regression, Naive Bayes, Decision Tree, and Support Vector Machine. There are three methods to establish a classifier: model a classification rule directly (e.g. Decision Tree), model the probability of class memberships given input data (e.g. Multilayer Perceptron), make a probabilistic model of data within each class (e.g. Naive Bayes). The first and the second methods are examples of discriminative classification. The second and the third methods are both examples of probabilistic classification. The third method is an example of generative classification.

4. Software quality model

4.1 Formulate the software quality model

In the absence of an agreed measure of software quality the number of software defects (e.g. number of software faults and number of software violations) has been a very commonly used surrogate measure. As a result, there have been numerous attempts to build models for predicting the number of software defects. Quality in a typical software artifact is a composite of quality-carrying attributes such as usability, portability, reliability, testability, reusability, and maintainability. As a result, we do not adopt a given model's characterization of quality in QUAMO. We propose a "define your own quality model" approach in QUAMO. In QUAMO, we need to formulate a composition in which we agree specific measures for the lowest-level attributes and specific relationships between the attributes apart from the Key Quality-carrying Attributes (KQA) described in Section 4.2 and Key Quality Metrics (KQM) described in Section 4.3. QUAMO is augmented from five software quality models: Boehm Model, McCall Model, FURPS, ISO 9126, and Dromey Model. In QUAMO, we measure the quality-carrying attributes objectively to investigate if the quality-carrying attributes meet the specified, quantified targets via different object-oriented metrics and Naive Bayes. QUAMO consists of 2 layers: the quality-carrying attribute layer and the object-oriented metrics layer. The upper branches hold important high-level quality-carrying attributes of object-oriented systems. Examples of such quality-carrying attributes are flexibility (i.e. to evaluate the effort required in modifying an operational class or component in an object-oriented system), maintainability (i.e. to evaluate the effort required in maintaining an operational class or component in an object-oriented system), reliability (i.e. to evaluate the extent to which an operational class or component performs its intended functional requirements in an object-oriented system), reusability (i.e. to evaluate the effort required in reusing an operational class or component in an object-oriented system), testability (i.e. to evaluate the effort required in testing an operational class or component in an object-oriented system), usability (i.e. to evaluate the effort required in learning and operating an operational class or component in an object-oriented system), and traceability (i.e. to evaluate the effort required in tracing an operational class or component in an object-oriented system). Each quality attribute is composed of lower-level criteria, namely object-oriented metrics (e.g. depth of inheritance tree, class size, and number of children). QUAMO generally resembles a tree that illustrates the important relationships between quality and its dependent criteria (i.e. quality-carrying attributes) so that quality in terms of the dependent criteria can be measured. Table 1 depicts a typical organization of input attributes, output attributes, and quality-carrying attributes in QUAMO. IA, IAV, OA, IAOAV, QCA, and IAQCAV collectively denotes input attribute, input attribute value (discrete value), output attribute, input attribute/output attribute

value (discrete value), quality-carrying attribute, input attribute/quality-carrying attribute value (discrete value). Examples of input attributes (i.e. effect/evidence) include KQM such as class inherited index and lack of class inherited index, and other object-oriented metrics. Examples of output attribute (i.e. cause) are number of software violations (i.e. compile-time defects) and number of software faults (i.e. run-time defects). Examples of quality-carrying attributes include KQA such as efficiency and reliability, and other quality-carrying attributes.

		OA_1	OA_2	…	OA_n	QCA_1	QCA_2	…	QCA_n
IA_1	IAV_1	IA_1OAV_1	IA_1OAV_2	…	IA_1OAV_n	IA_1QCAV_1	IA_1QCAV_2	…	IA_1QCAV_n
IA_2	IAV_2	IA_2OAV_2	IA_2OAV_2	…	IA_2OAV_n	IA_2QCAV_1	IA_2QCAV_2	…	IA_2QCAV_n
IA_3	IAV_3	IA_3OAV_3	IA_3OAV_2	…	IA_3OAV_n	IA_3QCAV_1	IA_3QCAV_2	…	IA_3QCAV_n
IA_4	IAV_4	IA_4OAV_4	IA_4OAV_2	…	IA_4OAV_n	IA_4QCAV_1	IA_4QCAV_2	…	IA_4QCAV_n
….	…	…	…	…	…	…	…	…	…
IA_n	IAV_n	IA_nOAV_n	IA_nOAV_2	…	IA_nOAV_n	IA_nQCAV_1	IA_nQCAV_2	…	IA_nQCAV_n

Table 1. Input attributes, output atributes, and quality-carrying attributes in QUAMO

4.2 Formulate the key quality-carrying attributes

KQA are quality-carrying attributes that present in all the five models studied in this research: Boehm Model, McCall Model, FURPS, ISO 9126, and Dromey Model. Table 2 depicts a comparison of the quality-carrying attributes in Boehm, McCall, FURPS, ISO 9126, and Dromey models. Since efficiency, reliability, and maintainability quality-carrying attributes present in all the models, they are considered essential in QUAMO. They are collectively referred to as KQA in QUAMO, which are mandatory attributes in QUAMO.

4.3 Formulate the key quality metrics

We propose eight KQM to measure the quality of object-oriented systems through QUAMO, namely Class Cohesion Index (CsCohI), Lack of Class Cohesion Index (LCsCohI), Component Cohesion Average (CoCohA), Lack of Component Cohesion Average (LCoCohA), Class Inherited Index (CsII), Lack of Class Inherited Index (LCsII), Component Inherited Average (CoIA), and Lack of Component Inherited Average (LCoIA). Table 3 and Table 4 depict the notations of CsCohI, CoCohA, LCsCohI, and LCoCohA, and the properties of CsCohI, CoCohA, LCsCohI, and LCoCohA, respectively. Similarly, Table 5 and Table 6 depict the notations of CsII, CoIA, LCsII, and LCoIA, and the properties of of CsII, CoIA, LCsII, and LCoIA, respectively.

4.3.1 Class-based cohesion metrics

We propose two class-based cohesion metrics, namely Class Cohesion Index (CsCohI) and Lack of Class Cohesion Index (LCsCohI) to measure the overall density of similarity and dissimilarity of methods in a class. CsCohI measures the degree of similarity of methods in a class. The CsCohI within a class Cs is expressed as:

$$\text{CsCohI(Cs)}=\begin{cases}\dfrac{\text{TCohM}}{\text{TM}}, \text{TCohM}>0 \text{ and TM} > 0\\ 0, \text{otherwise}\end{cases}$$

Each method within a class accesses one or more attributes (i.e. instance variables). CsCohI is the number of methods that access one or more of the same attributes. If no methods access at least one attribute, then CsCohI = 0. In general, low values for CsCohI imply that the class might be better designed by breaking it into two or more separate classes. Although there are cases in which a low value for CsCohI is justifiable, it is desirable to keep CsCohI high (i.e. keep cohesion high). CsCohI < 1 indicates that the class is not quite cohesive and may need to be refactored into two or more classes. Classes with a low CsCohI can be fault-prone. A low CsCohI value indicates scatter in the functionality provided by the class. CsCohI is expressed as a nondimensional value in the range of $0 \le \text{CsCohI} \le 1$. Similarly, the overall degree of dissimilarity of methods within a class Cs, LCsCohI(Cs), is expressed as:

$$\text{LCsCohI(Cs)}=\begin{cases}1-\dfrac{\text{TCohM}}{\text{TM}}, \text{TCohM}>0 \text{ and TM} > 0\\ 0, \text{otherwise}\end{cases}$$

4.3.2 Component-based cohesion metrics

We propose two component-based cohesion metrics, namely Component Cohesion Average (CoCohA) and Lack of Component Cohesion Average (LCoCohA) to measure the overall density of similarity and dissimilarity of methods in the classes within a component. CoCohA measures the degree of class cohesion indexes in a component. The CoCohA within a component Co is expressed as:

$$\text{CoCohA(Co)}=\begin{cases}\dfrac{\text{TCsCohI}}{\text{TC}}, \text{TCsCohI}>0 \text{ and TC} > 0\\ 0, \text{otherwise}\end{cases}$$

CoCohA is defined in an analogous manner and provides an indication of the overall degree of similarity of methods in the classes within a component. CoCohA is based on the notation that methods in the classes are similar if they share common instance variables. The larger the number of similar methods in the classes within a component, the more cohesive the component. Hence, CoCohA is a measure of the relatively disparate nature of the methods in the classes within a component. The CoCohA numerator is the sum of class cohesion indexes in a component, TCsCohI. The CoCohA denominator is the total classes in a component. The CoCohA numerator represents the maximum number of similarity of method situations in the classes for a component. CoCohA is expressed as a nondimensional value in the range of $0 \le \text{CoCohA} \le 1$. In general, a low value for CoCohA indicates a low proportion of class cohesion indexes in a component, and a high value for CoIA indicates a high proportion of class cohesion indexes in a component. A low value for CoCohA is undesirable. Similarly, the overall degree of dissimilarity of methods in the classes within a component Co, LCoCohA(Co), is expressed as:

$$\text{LCoCohA(Co)}=\begin{cases}1-\dfrac{\text{TCsCohI}}{\text{TC}}, \text{TCsCohI} > 0 \text{ and TC} > 0\\ 0, \text{otherwise}\end{cases}$$

Quality-carrying Attributes	Software Quality Models				
	Boehm Model (1978)	McCall Model (1977)	FURPS (1987)	ISO 9126 (1991)	Dromey Model (1995)
Testability	x	x		x	
Correctness		x			
Efficiency	x	x	x	x	x
Understandability	x			x	
Reliability	x	x	x	x	x
Flexibility		x	x		
Functionality			x	x	x
Human Engineering	x				
Integrity		x		x	
Interoperability		x		x	
Process Maturity					x
Maintainability	x	x	x	x	x
Changeability	x				
Portability	x	x		x	x
Reusability		x			x

Table 2. Quality-carrying attributes in Boehm model, McCall model, FURPS, ISO 9126, and Dromey model

4.3.3 Discussions

LCsCohI and LCoCohA are inverse metrics of CsCohI and CoCohA respectively. A high value of CsCohI, and CoCohA, and a low value of LCsCohI and LCoCohA indicate high cohesion and well-designed class and component. Similarly, a low value of CsCohI, and CoCohA, and a high value of LCsCohI and LCoCohA indicate low cohesion and poorly designed class and component. It is likely that the class and component have good subdivision. A cohesive class tends to provide a high degree of encapsulation. A lower value of CsCohI and CoCohA indicate decreased encapsulation, thereby increasing the likelihood of errors. Similarly, a lower value of LCsCohI and LCoCohA indicate increased encapsulation, thereby decreasing the likelihood of errors.

4.3.4 Class-based inheritance metrics

We propose two class-based inheritance metrics, namely Class Inherited Index (CsII) and Lack of Class Inherited Index (LCsII) to measure the overall inheritance density in a class.

CsII measures the degree of inherited attributes and methods in a class. The CsII within a class Cs is expressed as:

$$\text{CsII(Cs)}=\begin{cases}\dfrac{\text{TIA+TIM}}{\text{TM+TA}}, & \text{TIA+TIM}>0 \text{ and } \text{TM}+\text{TA} > 0\\ 0, & \text{otherwise}\end{cases}$$

CsII is defined in an analogous manner and provides an indication of the impact of inheritance at the class level. The CsII numerator is the sum of inherited attributes and methods in a class. The CsII demoninator is the total number of attributes and methods in a class. The CsII numerator represents the maximum number of possible distinct inheritance situations for a class. CsII is expressed as a nondimensional value in the range of $0 \le \text{CsII} \le 1$. In general, a low value for CsII indicates a low proportion of inherited attributes and methods in a class, and a high value for CsII indicates a high proportion of inherited attributes and methods in a class. A high value of CsII is undesirable. As the number of inherited attributes and methods increases, the value of CsII also increases. Similarly, the overall degree of non-inherited attributes and non-inherited methods within a class Cs, LCsII(Cs), is expressed as:

$$\text{LCsII(Cs)}=\begin{cases}1-\dfrac{\text{TIA+TIM}}{\text{TM+TA}}, & \text{TIA+TIM} > 0 \text{ and } \text{TM+TA} > 0\\ 0, & \text{otherwise}\end{cases}$$

4.3.5 Component-based inheritance metrics

We propose two component-based inheritance metrics, namely Component Inherited Average (CoIA) and Lack of Component Inherited Average (LCoIA) to measure the overall inheritance density in the classes of within a component. CoIA measures the degree of class inherited indexes in a component. The CoIA within a component Co is expressed as:

$$\text{CoIA(Co)}=\begin{cases}\dfrac{\text{TCsII}}{\text{TC}}, & \text{TCsCII}>0 \text{ and } \text{TC} > 0\\ 0, & \text{otherwise}\end{cases}$$

CoIA is defined in an analogous manner and provides an indication of the impact of inheritance at the component level. The CoIA numerator is the sum of class inherited indexes in a component. The CoIA denominator is the total classes in a component. The CoIA numerator represents the maximum number of possible distinct inheritance situations for a component. CoIA is expressed as a nondimensional value in the range of $0 \le \text{CoIA} \le 1$.
In general, a low value for CoIA indicates a low proportion of class inherited indexes in a component, and a high value for CoIA indicates a high proportion of class inherited indexes in a component. A high value for CoIA is undesirable. As the class inherited indexes increases, the value of CoIA also increases. Similarly, the overall degree of non-inherited attributes and non-inherited methods in the classes within a component Co, LCoIA(Co), is expressed as:

$$\text{LCoIA(Co)}=\begin{cases}1-\dfrac{\text{TCsII}}{\text{TC}}, & \text{TCsCII} > 0 \text{ and } \text{TC} > 0\\ 0, & \text{otherwise}\end{cases}$$

Notation	Description	C++	Java
CsCohI	class cohesion index within a class	-	-
CoCohA	component cohesion average within a component	-	-
LCsCohI	lack of class cohesion index within a class	-	-
LCoCohA	lack of component cohesion average within a component	-	-
TC	total number of classes in a component	total number of classes in a directive	total number of classes in a package
TCsCohI	total of class cohesion indexes in a component	-	-
TCohM	methods declared and inherited in a class assessing at least one instance variable	all function members declared and inherited in a class excluding virtual (deferred) ones assessing at least one instance variable	all methods declared and inherited in a class excluding abstract (deferred) ones assessing at least one instance variable
TM	methods declared and inherited in a class	all function members declared and inherited in a class excluding virtual (deferred) ones	all methods declared and inherited in a class excluding abstract (deferred) ones

Table 3. Notations of CsCohI, CoCohA, LCsCohI, and LCoCohA

4.3.6 Discussions

LCsII and LCoIA are inverse metrics of CsII and CoIA respectively. A high value of CsII, and CoIA, and a low value of LCsII and LCoIA indicate high inheritance. Similarly, a low value of CsII, and CoIA, and a high value of LCsII and LCoIA indicate low inheritance. A lower value of CsII and CoIA indicate decreased inheritance and complexity, thereby decreasing the likelihood of errors. Similarly, a higher value of LCsII and LCoIA indicate increased inheritance and complexity, thereby increasing the likelihood of errors.

4.4 Formulate the software quality prediction model

This research adopts supervised learning through Naive Bayes to formulate the software quality prediction model. The primary objective of adopting supervised learning in QUAMO is to infer a functional mapping based on a set of training examples to assess the quality of object-oriented code. More specifically, the supervised learning in QUAMO can be formulated as the problem of inferring a function $y = f(x)$ based on a training set $D = \{(x_1,$

y_1), {(x_2, y_2), {(x_3, y_3), {(x_4, y_4),…, (x_n, y_n)}. The obtained function is evaluated by how well it generalizes. This research uses Naive Bayes as the primary method to predict software quality. Naive Bayes classifiers can be trained very efficiently in a supervised learning.

Properties for CsCohI and LCsCohI	
Property 1	If n is a non-negative number, then there is only a finite number of cohesive methods (i.e. number of methods assessing at least one instance variable) TCohM for which TCohM = n.
Property 2	If n is a non-negative number, then there is only a finite number of attributes declared and inherited in a class TCsA for which TCsA = n.
Property 3	If n is a non-negative number, then there is only a finite number of methods declared and inherited in a class TCsM for which TCsM = n.
Property 4	There are distinct classes Cs1 and Cs2 for which Cs1 is the superclass of Cs2.
Property 5	There are inherited attributes, IA1, IA2, IA3,…, IAn, for which IA1 ≠ IA2 ≠ IA3 … ≠ IAn
Property 6	There are inherited methods, IM1, IM2, IM3,…, IMn, for which IM1 ≠ IM2 ≠ IM3 … ≠ IMn
Properties for CoCohA and LCoCohA	
Property 1	If n is a non-negative number, then there is only a finite number of classes TCs for which TCs = n.
Property 2	If n is a decimal number, then there are total class inherited indexes TCsCohI for which TCsCohI = n.

Table 4. Properties of CsCohI, CoCohA, LCsCohI, and LCoCohA

Naive Bayes also generally gives better test accuracy than any other know machine learning techniques such as ID3 Decision Tree, C4.5 Decision Tree, and J48 Decision Tree. We can greatly simplify learning in software quality prediction by assuming that quality-carrying features are independent of each other through Naive Bayes. Naive Bayes assumes that the presence or absence of a particular feature of a class is unrelated to the presence or absence of any other feature. Naive Bayes learning gives better test set accuracy than any other known method, including Backpropagation and Decision Trees. Naive Bayes classifier can also be learned very efficiently. We have selected Naive Bayes as the primary technique to assess software quality in object-oriented code through a 2-layer "define your own quality model" based on a suite of object-oriented metrics.

The Naive Bayes classifier in QUAMO learns the conditional probability of each quality-carrying attribute QA_i given the class label C (i.e. a discretized value of an object-oriented metric). Classification is performed by applying Bayes rule to compute the probability of C given the particular instance of QA_1, QA_2, QA_3, QA_4, QA_5, …, QA_n, and then predicting the class with the highest posterior probability. This computation is possible by making a strong independence assumption that all the quality attributes QA_i are

conditionally independent given the value of the class C. We refer independence as probabilistic independence (i.e. X is independent of Y given Z when P(X | Y, Z) = P(X | Z) for all possible values of X, Y, and Z when P(Z) > 0. When X is a vector of discrete-valued object-oriented metrics (e.g. binary, X ∈ {low, high}), we adopt a 2-step approach: learn

Notation	**Description**	**C++**	**Java**
CsII	class inherited index	-	-
LCsII	lack of class inherited index	-	-
CoIA	component inherited average	-	-
LCoIA	lack of component inherited average	-	-
TC	total classes	total number of classes in a directive	total number of classes in a package
TCsII	total class inherited indexes	-	-
TM	Methods declared and inherited	all function members declared and inherited in a class including virtual (deferred) ones	all methods declared and inherited in a class including abstract (deferred) ones
TA	attributes declared and inherited	all data members declared and inherited in a class	all attributes declared and inherited in a class
TIM	methods inherited	all function members inherited and not overridden	all methods inherited in a class and not overridden
TIA	attributes inherited	all data members inherited in the class	all attributes inherited in a class

Table 5. Notations of CsII, CoIA, LCsII, and LcoIA

and test. When X is a vector of continuous-valued object-oriented metrics, we adopt a 3-step approach: discretize, learn, and test. Figure 1 depicts a typical Naive Bayes classifier in QUAMO. We can view the function approximation learning algorithm adopted in QUAMO as statistical estimators of conditional distributions P(Y | X) or of functions that estimate P(Y

| X) from a sample of training data in QUAMO. Naive Bayes uses Bayes' Theorem to predict the value of a target (i.e. output in QUAMO), from evidence given by one or more predictor (i.e. input in QUAMO) fields. Table 7, Table 8, and Table 9 depict the discretize, learn, and test algorithms.

Properties for CsII and LcsII	
Property 1	If n is a non-negative number, then there is only a finite number of inherited attributes TCsIA for which TCsIA = n.
Property 2	If n is a non-negative number, then there is only a finite number of inherited methods TCsIM for which TCsIM = n.
Property 3	If n is a non-negative number, then there is only a finite number of attributes TCsA for which TCsA = n.
Property 4	If n is a non-negative number, then there is only a finite number of methods TCsM for which TCsM = n.
Property 5	There are distinct classes Cs1 and Cs2 for which Cs1 is the superclass of Cs2.
Property 6	There are inherited attributes, IA1, IA2, IA3,…, IAn, for which IA1 ≠ IA2 ≠ IA3 … ≠ IAn
Property 7	There are inherited methods, IM1, IM2, IM3,…, IMn, for which IM1 ≠ IM2 ≠ IM3 … ≠ IMn
Properties for CoIA and LCoIA	
Property 1	If n is a non-negative number, then there is only a finite number of classes TCs for which TCs = n.
Property 2	If n is a decimal number, then there are total class inherited indexes TCsII for which TCsII = n.

Table 6. Properties of CsII, CoIA, LCsII, and LcoIA

5. Conclusion

The primary objective of this research is to propose the characteristics of a quality model through a comparative evaluation of existing software quality models. Based on the

comparative evaluation, an improved hierarchical model, QUAMO for the assessment of high-level quality attributes in object-oriented systems specializing on object-oriented code based on object-oriented metrics and Naive Bayes is proposed. In this model, the structural properties of classes and their relationships are evaluated using Naive Bayes and a suite of object-oriented metrics. A key attribute of QUAMO is that the model can be augmented to include different object-oriented metrics and quality-carrying attributes, thus providing a practical quality assessment instrument adaptable to a variety of object-oriented systems. QUAMO relates code properties (also referred to as object-oriented constructs) such as encapsulation, information hiding, and inheritance to high-level quality carrying attributes such as reusability, flexibility, maintainability, and complexity via Naive Bayes and a suite of object-oriented metrics.

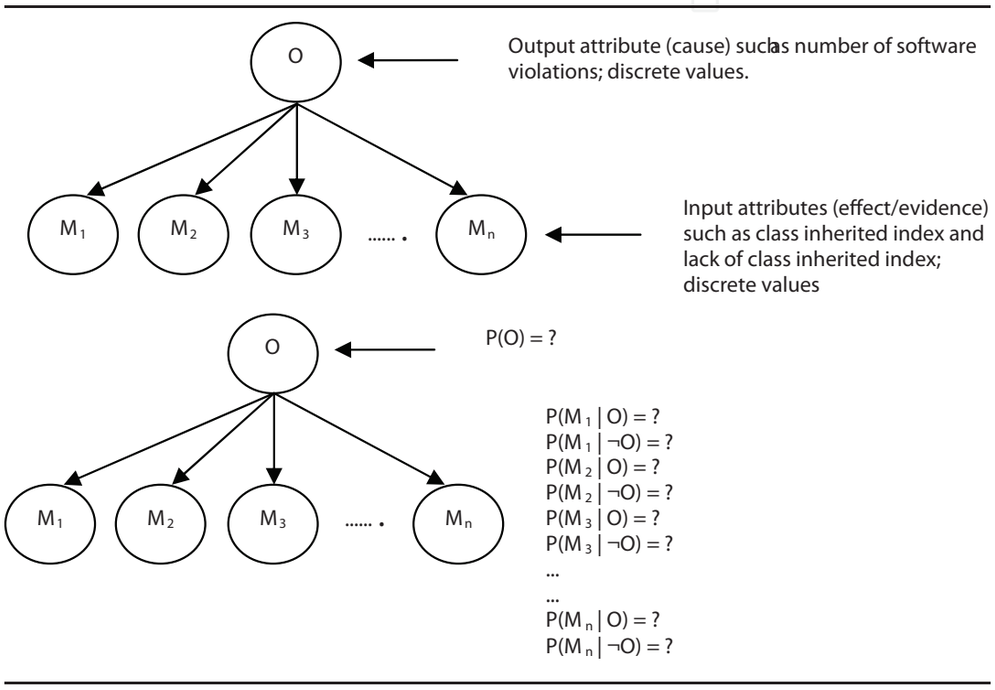


Fig. 1. QUAMO Naive Bayes Classifier

Precondition: There are n training instances for which the value of a numeric attribute (e.g. KQM and other object-oriented metrics, and outputs such as number of software violations and number of software bugs) χ_i is known. The minimum and maximum values are v_{min} and v_{max} respectively.
Postcondition: There are k intervals for which the width $w = (v_{max} - v_{min} / k)$.
Rule: The values of the metrics are continuous values.

Algorithm:
Given a numeric attribute χ_i
Sort the values of v_i ($_{vi} = v_1, ..., v_n$) in ascending order
Divide the sorted values of v_i between v_{min} and v_{max} into intervals of equal width

Table 7. Discretize Algorithm

Postcondition: Conditional probability tables for x_j, N_j x L elements.
Rule: The values of the attrtibutes values are discrete values and the values of target values are continuous values.

Algorithm:
Given a training set S
For each target value of c_i ($c_i = c_1, ..., c_L$)
    $P'(C = c_i) \leftarrow$ estimate $P(C = c_i)$
    For every attribute value a_{jk} of each attribute x_j ($j = 1, ..., n$; $k = 1, ..., N_j$)
$P'(X_j = a_{jk} \mid C = c_i) \leftarrow$ estimate $P(X_j = a_{jk} \mid C = c_i)$

Table 8. Learn Algorithm

Precondition: Conditional probability tables for for x_j, N_j x L elements.
Postcondition: c or c^* is labelled to X'.
Rule: None.

Algorithm:
Given an unknown instance $X' = (a'_1, ..., a'_n)$
Look up conditional probability tables to assign the label c^* to X'
    Compute $[P'(a'_1 \mid c^*) ... P'(a'_n \mid c^*)]P'(c^*)$
    Compute $[P'(a'_1 \mid c^*) ... P'(a'_n \mid c^*)]P'(c)$
    If $[P'(a'_1 \mid c^*) ... P'(a'_n \mid c^*)]P'(c^*) > [P'(a'_1 \mid c^*) ... P'(a'_n \mid c^*)]P'(c)$,
        $c \neq c^*$, $c = c_1, ..., c_L$ then
        Label X' to be c^*
    Else
        Label X' to be c

Table 9. Test Algorithm

6. References

Bansiya, J. & Davis, C.G. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, Vol.28, No.1, pp. 4-17.

Boehm, B. W.; Brownm, J. R. & Lipow M. (1976). Quantitative evaluation of software quality, *Proceedings of the 2nd International Conference on Software engineering*, pp. 592-605, San Francisco, California, United States.

Callaos, N. & Callaos, B. (1996) . Designing with a systemic total quality, *Proceedings of the International Conference on Information Systems Analysis and Synthesis*, pp. 15-23, Orlando, Florida, United States.

Chidamber S.R. & Kemerer C.F. (1994). A metrcs suite for object-oriented design. *IEEE Transactions on Software Engineering*, Vol.20, No.6, pp. 476-493.Côté, M.A.; Suryn, W. & Georgiadou, E. (2007). In search for a widely applicable and accepted software quality model for software quality engineering. *Software Quality Journal*, Vol.15, No.4, pp. 401-416.

Dromey, R.G. (1995). A model for software product quality. *IEEE Transactions on Software Engineering*, Vol.21, No.1, pp. 146-162.

Dromey, R.G. (1996). Concerning the Chimera [software quality]. *IEEE Transactions on Software Engineering*, Vol.13, No.1, pp. 33-43.

Georgiadou, E. (2003). GEQUAMO – A generic, multilayered, customizable, software quality model. *Software Quality Control*, Vol.11, No.4, pp. 313-323.

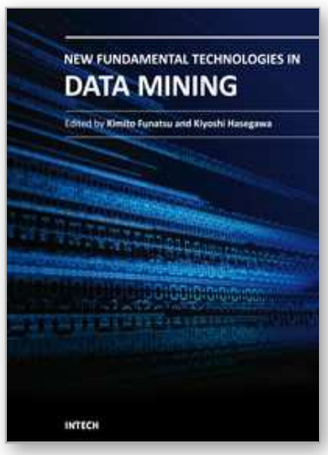
Khaddaj, S. & Horgan, G. (2005). A proposed adaptable quality model for software quality assurance, *Journal of Computer Science*, Vol.1, No.4, pp. 482-487.

McCall, J.A., Richards, P.K. & Walters, G.F. (1977). Factors in software quality. *National Technical Information Service*, Vol.1-3.

Ortega, M.; Pérez, M. & Rojas, T. (2000). A model for software quality with a systemic focus, *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics*, pp. 464-469, Orlando, Florida, United States.

Ortega, M.; Pérez, M. & Rojas, T. (2002). A systemic quality model for evaluating software products, *Proceedings of the 6th World Multiconference on Systemics, Cybernetics and Informatics*, pp. 371-376, Orlando, Florida, United States.

Ortega, M.; Pérez, M. & Rojas, T. (2003). Construction of a systemic quality model for evaluating a software product. *Software Quality Journal*, Vol.11, No.3, pp. 219-242.



New Fundamental Technologies in Data Mining

Edited by Prof. Kimito Funatsu

ISBN 978-953-307-547-1
Hard cover, 584 pages
Publisher InTech
Published online 21, January, 2011
Published in print edition January, 2011

The progress of data mining technology and large public popularity establish a need for a comprehensive text on the subject. The series of books entitled by "Data Mining" address the need by presenting in-depth description of novel mining algorithms and many useful applications. In addition to understanding each section deeply, the two books present useful hints and strategies to solving problems in the following chapters. The contributing authors have highlighted many future research directions that will foster multi-disciplinary collaborations and hence will lead to significant development in the field of data mining.

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen