Predicting Software Suitability Using a Bayesian Belief Network

Justin M. Beaver
NASA, Kennedy Space Center
Justin.M.Beaver@nasa.gov

Guy A. Schiavone
University of Central Florida
Guy@cs.ucf.edu

Joseph S. Berrios
University of Central Florida
JBerrios@mail.ucf.edu

Abstract

The ability to reliably predict the end quality of software under development presents a significant advantage for a development team. It provides an opportunity to address high risk components earlier in the development life cycle, when their impact is minimized. This research proposes a model that captures the evolution of the quality of a software product, and provides reliable forecasts of the end quality of the software being developed in terms of product suitability. Development team skill, software process maturity, and software problem complexity are hypothesized as driving factors of software product quality. The cause-effect relationships between these factors and the elements of software suitability are modeled using Bayesian Belief Networks, a machine learning method. This research presents a Bayesian Network for software quality, and the techniques used to quantify the factors that influence and represent software quality. The developed model is found to be effective in predicting the end product quality of small-scale software development efforts.

1. Introduction

Software quality is perhaps the most vaguely defined and overused term in the field of software engineering. It is meant to encompass all of the stakeholder needs and perspectives in terms of the delivered software product [1]. It is meant to include both objective and subjective technical evaluations. "High" quality is the inherent, yet vastly subjective goal of every software development team. A standardized approach to quantifying both the elements of software quality, and the factors that influence software quality is essential to providing insight into a software product that is consistent and reliable across applications.

Modeling software quality involves both a current assessment of the software development effort, and a prediction of the quality of the delivered software product. Accuracy in modeling the complexities of a software development effort is essential to providing meaningful insight into the predicted quality of the software product at the time of delivery. Such insight allows a development team to identify and address problem areas within an evolving software system. Historically, the quality models developed in the software engineering community are suspect largely because of their inability to be universally applicable, and their inherent data quality problems [2]. The contributions to software quality modeling that focus on using complex adaptive systems have shown potential in accurately representing cause-effect relationships in software product development, and providing models that are adaptable to a given development team and environment, and perform well in the presence of uncertain or incomplete data [3] [4] [5].

This research effort addresses a need for improved insight into the quality of software under development. The product of this research is a software quality model that attempts to provide a reliable projection of the software product quality using various measures from the software development life cycle.

2. Software Quality Modeling Approach

What causes software quality in a product? There are many examples in software engineering literature [9] [10] [11] in which empirical relationships are established between software development measures and software quality measures. Trends among various measurements are identified, but the issue of causality is insufficiently addressed [2]. That is, the establishment of an empirical relationship between a design metric and a software quality measure does not prove, or for that matter even imply, that the measured design characteristic caused the software quality characteristic. Empirical relationships must be enhanced with logic in order to be considered as representing cause-effect [6]. Consider a hypothetical study that identifies an empirical relationship between the number of lines-of-code in a software component and the corresponding number of defects. Would a viable approach to reducing defects be to consolidate several lines into a single statement, thus reducing the component size? Does this improve the quality of the software product? Such a study may have executed a perfectly correct statistical experiment, and yet has provided little value to the software engineering community.

This software quality research effort attempts to identify, model, and validate those factors that influence the quality of a software product. The following three premises are proposed as a basis for the structure of the software quality model:

1. Maturity of software development processes is a causal factor in software product quality.
2. Complexity of the software problem is a causal factor in software product quality.
3. Capability of the software development team is a causal factor in software product quality.

The intent of this research is to model these causal factors as drivers of software product quality, and validate the accuracy of the model in predicting product quality.
Figure 1 below is a cause-effect diagram that details the way in which the three premises above are used to predict specific software quality values.

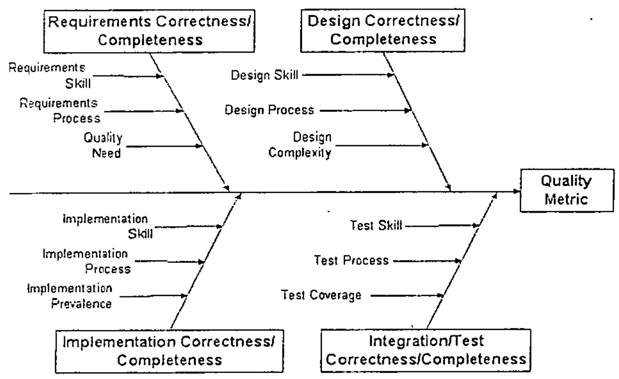
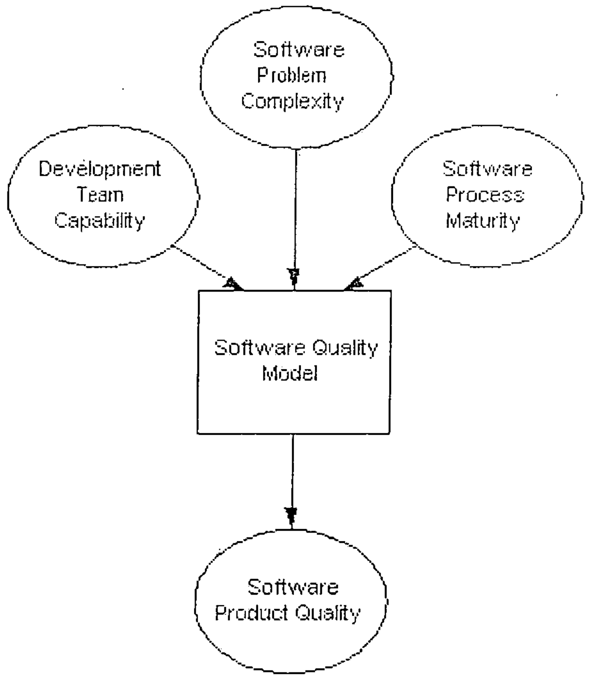


Figure 1. Causal Factors of a Software Quality Metric

Quality metrics are affected by the correctness and completeness of the activities and artifacts at each phase of the development life cycle. Measures of correctness and completeness are driven by the development team's skill/experience level, the process maturity, and the complexity for each phase

The proposed quality model defines software development cause-effect relationships in the context of a Bayesian Belief Network (BBN). A BBN is a directed acyclic graph containing nodes and arcs. The nodes represent discrete random variables within the model, and the arcs represent the cause-effect relationship between the nodes. The network is called Bayesian because the transformation function from inputs to outputs at each node is based on Bayes' Rule for conditional probability. Each node is characterized by a conditional probability table, which contains the probabilities of each possible output state in terms of the possible combinations of input states. BBNs learn their prior probabilities for the various state combinations of the inputs and outputs using prior data sets. The adaptive nature of the BBN makes it an attractive option for representing software quality accurately at the local level. That is, while the model structure is universal, the application of that structure is specific to a given organization and dependent on the data that populates their local conditional probability tables.



Software Problem Complexity
Development Team Capability
Software Process Maturity
Software Quality Model
Software Product Quality

Figure 2. High-Level Software Quality Model

Figure 2 is a high level depiction of the structure of the Bayesian Belief Network used to model software product quality. Inputs to the model are ratings of the development team's capability, maturity of the project's development process, and complexity of the software problem. The output of the model is calculated values of software product quality. The software quality model itself relates the inputs and outputs through a set of intermediary nodes that represent the correctness and completeness at each phase of the development life cycle. The internal structure of the software quality model is addressed in more detail in Section 4.

3. Quantifying Software Quality Factors

In order to effectively model the quality of a software product, it is necessary to quantify all aspects of the inputs and outputs of the proposed model in terms of usable measures. The following paragraphs detail the approach taken to quantify the various aspects of development team capability, software process maturity, software problem complexity, and software product quality.

3.1 Assessing Development Team Capability

The approach taken to assess the development team capability involved two stages: to provide a criterion for rating the skills of individual team members, and then to provide a method for consolidating the various individual capabilities into a team capability. At the individual level, a four-tier ordinal rating scheme was created to capture a software developer's skill and experience. Each software project team member is assigned a separate rating (1, 2, 3, or 4) for each of the four major life cycle phases: Requirements Development, Design Development, Implementation, and Integration/Test. The rating of an individual for a given phase is based on a set of criteria for that phase that takes formal education, formal training, and industry experience into account. Thus, each project team member will have four ratings, one for each of the four phases, and each rating is on an ordinal scale (1-4) based on the individual's education and experience.

The overall Development Team Capability is captured as 16 discrete variables which represent the distribution of team skill in each of the four life cycle phases. Table 1 below lists those variables that were used to quantify the collective skill and experience of a software development team. Each of the 16 variables is comprised of a four-tiered scale representing the percentage of the team with the corresponding skill level. Each tier corresponds to a percentage range of the overall team. For example, one of the 16 variables used to represent overall Development Team Capability captures the proportion of team members with Requirements Skill Level 1. The four tiers used represent the ranges 0%-25%, 26%-50%, 51%-75%, and 76%-100%. So, if there are four development team members involved in requirements development, and one of them has been appraised at Requirements Skill Level 1, then the Requirements Skill 1 rating for the overall team is 1 (25% of the team). Similarly, proportions may be determined for each of the 16 team skill ratings (4 skill levels for each of the four phases). This level of detail provides the model with a complete picture the skill set within the development team.

Table 1. Development Team Capability Factors

Variable Name	Description
Requirements Level 1	Percentage of requirements team assessed at skill/experience level 1.
Requirements Level 2	Percentage of requirements team assessed at skill/experience level 2.
Requirements Level 3	Percentage of requirements team assessed at skill/experience level 3.
Requirements Level 4	Percentage of requirements team assessed at skill/experience level 4.
Design Level 1	Percentage of design team assessed at skill/experience level 1.
Design Level 2	Percentage of design team assessed at skill/experience level 2.
Design Level 3	Percentage of design team assessed at skill/experience level 3.
Design Level 4	Percentage of design team assessed at skill/experience level 4.
Implementation Level 1	Percentage of implementation team assessed at skill/experience level 1.
Implementation Level 2	Percentage of implementation team assessed at skill/experience level 2.
Implementation Level 3	Percentage of implementation team assessed at skill/experience level 3.
Implementation Level 4	Percentage of implementation team assessed at skill/experience level 4.
Test Level 1	Percentage of integration/test team assessed at skill/experience level 1.
Test Level 2	Percentage of integration/test team assessed at skill/experience level 2.
Test Level 3	Percentage of integration/test team assessed at skill/experience level 3.
Test Level 4	Percentage of integration/test team assessed at skill/experience level 4.

3.2 Assessing Software Process Maturity

Software process maturity is quantified using the ISO/IEC 15504 [7] standard as guide in determining the elements of the software process to consider. The ISO/IEC 15504 is an international standard for software process assessment, and is the product of a collaborative effort of several major software process improvement efforts. The ISO/IEC 15504 is comprised of four different process categories that address four distinct areas within a software development project: project management processes, engineering processes, support processes, and customer-supplier processes. As this research focuses solely on the technical quality of the software product, the Engineering process category, which is the set of processes that cover the specification, design, implementation and integration/test of the software product will be the area that will be considered as relevant to the model. The Engineering process category of the ISO/IEC 15504 standard consists of seven different processes, five of which pertain to the software engineering life cycle.

Table 2. Software Process Maturity Factors

ISO/IEC 15504 Engineering Process Category	
Process	Practice
Software Requirements Analysis	Specify Software Requirements
	Determine Operating Environment Impact
	Evaluate/Validate Requirements with Customer
	Develop Validation Criteria for Software
	Develop Release Strategy
	Update Requirements
	Communicate Software Requirements
	Evaluate the Software Requirements
Software Design	Develop Software Architectural Design
	Design Interfaces
	Verify the Software Design
	Develop Detailed Design
	Establish Traceability
Software Construction	Develop Software Units
	Develop Unit Verification Procedures
	Verify the Software Units
	Establish Traceability
Software Integration	Develop Software Integration Strategy
	Develop Integrated Software Item Regression Strategy
	Develop Tests for Integrated Software Items
	Test Integrated Software Items
	Integrate Software Item
	Regression Test Integrated Software Items
Software Testing	Develop Integrated Software Test Strategy
	Develop Tests for Integrated Software
	Test Integrated Software
	Regression Test Integrated Software

Categorized in the five software engineering processes are 27 practices (See Table 2) that identify the activities necessary to develop software in a consistent manner and with repeatable results. Each software project used for this research was evaluated in terms of these practices. A binary (yes/no) indicator of compliance with each practice was used to quantify its software process maturity.

3.3 Assessing Software Problem Complexity

Software problem complexity is captured differently for each phase of the development life cycle. In the requirements phase, complexity is determined by assessing whether or not the functional needs of the customer have been addressed, and how frequently those needs change after baseline. In the design phase, the complexity of the software problem can be characterized by the complexity of the design itself and the volatility of the design baseline. Complexity in the implementation phase is represented by the prevalence of quality requirements and needs in the software, and by the volatility of the source code units. Test complexity represents the extent to which the test covers the expected functionality of the product.

Table 3. Software Problem Complexity Factors

Life Cycle Phase	Software Complexity Indicator
Requirements	Has the expected functional operation of the software been described?
	How volatile are the requirements?
Design	How complex is the design of the software?
	How complex is the design of the interfaces?
	How volatile is the design?
Implementation	How complex is the implemented software?
	How prevalent are quality needs in the implemented software?
	How volatile is the implementation?
Integration/Test	How well has the test covered the expected functionality?

Table 3 summarizes the indicators used to represent the complexity of the software problem. This approach captures the breadth of expectations of the software product in terms of an accepted standard for software quality, and provides a quantification of the nature of the problem that is being solved with the development of the software product. Once the expectations for a project have been established, the fulfillment of those expectations may also be measured using the same standard.

3.4 Assessing Software Product Quality

Software product quality in this research is captured using the ISO/IEC 9126 [11] as a guide for the expectations of quality within the delivered software product. The ISO/IEC 9126 is an international standard for software product quality that represents the quality of a delivered software product in terms of six major characteristics: Functionality, Efficiency, Reliability, Usability, Maintainability, and Portability. In the standard, each of the six quality characteristics is further partitioned into sub-characteristics and associated indicator metrics that allow for consistent measurement and assessment of quality.

Table 4. Software Product Quality Measures

Metric Name	Metric Description
Functional Adequacy	Number of functional requirements present and correctly implemented in the software product.
Functional Implementation Completeness	Number of functional requirements present in the software product.
Functional Implementation Coverage	Number of functional requirements correctly implemented in the software product.
Functional Specification Stability	Number of functional requirements unchanged after requirements baseline.

This research has focused on capturing and modeling product quality in terms of the Suitability of the product. Suitability refers to the adequacy of the software product in terms of its coverage of user needs and correctness of implementation. Table 4 lists the metrics used to capture software product quality for the Suitability portion of the ISO/IEC 9126 standard. All of the software product quality metrics are represented in terms of the number of needs or requirements that were verified to be correctly implemented. This implies correctness and completeness in the capture of the customer's needs during all of the life cycle phases.

4. Software Quality Model Structure

Modeling the various quality metrics was accomplished by representing the cause-effect diagram shown in Figure 1 in a Bayesian Belief Network. The same general approach was used for each of the software quality metrics. For each quality metric, the structure of the model attempts to take into account the correctness and completeness of the activities associated with each of the four software development life cycle phases. Figure 3 below describes the approach in terms of a BBN.

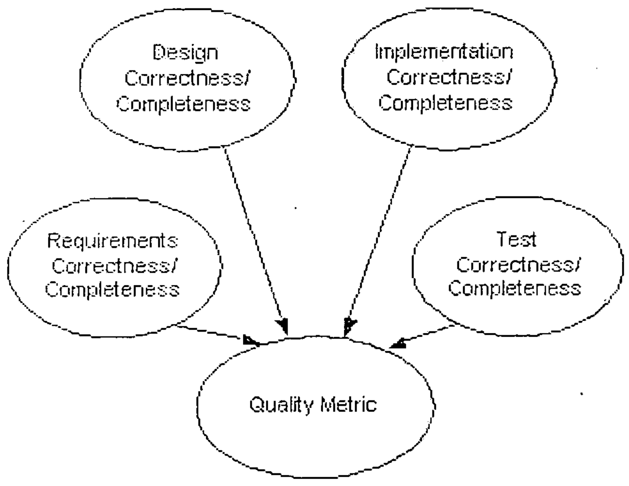

Design Correctness/ Completeness
Implementation Correctness/ Completeness
Requirements Correctness/ Completeness
Test Correctness/ Completeness
Quality Metric

Figure 3. Bayesian Network for Modeling a Quality Metric

These indicators of correctness and completeness are fed by the drivers of software quality: development team capability, process maturity, and problem complexity. Figure 4 depicts the generic model structure that is applied for each software product quality indicator. The correctness of each phase is represented by variables that indicate the amount of change associated with the phase artifacts, and the proportion of requirements, design modules, etc. that were verified to have been implemented correctly. The completeness of each phase is represented by variables that indicate the degree to which the user needs, requirements, or design was addressed in the phase.

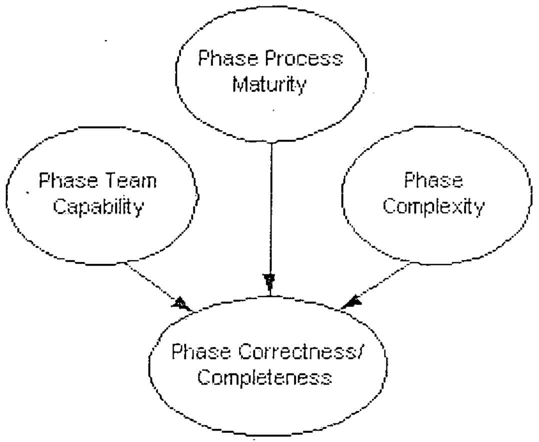


Figure 4. Bayesian Network for Modeling Phase Correctness and Completeness

5. Software Quality Model Validation

The approach taken to validate the software quality model is to make statistical inferences about the accuracy of the model when making predictions about software product quality.

5.1 Data Description

The sets of data used to train and validate this software quality model are from 21 software development projects. The projects were small in scale, and were generally completed within a 3-4 month time frame. Project teams varied in size from 1 to 4 developers and included both graduate students and software engineering professionals. Projects were required to sequentially address each phase of the development life cycle in a classic "waterfall" fashion.

Each project was asked to track various software engineering metrics through the development life cycle. These metric were reported at the conclusion of each phase. Of the original 28 projects selected to participate in this research, 21 were actually used because of their willingness to track life cycle measures completely and correctly.

5.2 Validation Approach

The proposed software quality model was validated in terms of its predictive accuracy. The approach to validation was to evaluate the predicted values of the selected software quality metrics at the conclusion of the software design phase with the actual values of those metrics at delivery of the software product. A validation run was performed for each of the 21 participating projects. In each validation run, the model was trained using all but one of the available data sets. The unused data set served as the data set for which the model made predictions about the various software quality measures. The predictions were recorded along with the actual software quality values for that data set. This process was repeated for each of the available data sets. The resultant data is a collection of expected versus actual values for each of the software quality measures being validated.

The statistical method for validation is a hypothesis test for equality of means between expected and actual values for each software quality metric. In this case, the null hypothesis is that the means are equal, and the alternative hypothesis is that they are different. The decision rule for the hypothesis test is shown below:

H_0: $\mu_{\text{modeled}} - \mu_{\text{actual}} = 0$
H_a: $\mu_{\text{modeled}} - \mu_{\text{actual}} \neq 0$

Reject H_0 if:

$$\left| \frac{\mu_{actual} - \mu_{modeled}}{\sqrt{msE * (2/n)}} \right| < t_{n-\upsilon,\alpha/2}$$

where,

μ_{modeled}	= the mean value of the modeled variable
μ_{actual}	= the mean value of the actual variable
msE	= the mean square error
n	= the total number of samples
v	= the number of degrees of freedom
$t_{n-v,\alpha/2}$	= t-distribution for confidence $(1-\alpha)100\%$

If the test statistic for the given quality measure exceeds the t-distribution value for that quality measure, then the null hypothesis must be rejected. For this study, a confidence of $\alpha = 0.9$, or 90% was used for all statistical calculations. In addition to the contrasted means, the sample size required to make an inference on the contrasted means is reported. This insures that the sample sizes are sufficient to make a statistical inference.

5.3 Results

The developed software quality model performed well in predicting for the four measures of suitability. Table 5 contains the results from applying the hypothesis test to determine whether or not the modeled values for the software suitability metrics assumed the same distribution as the actual values for those metrics. For all of the metrics, the sample sizes were found to be sufficient when compared to the sample sizes needed for significance. The means and variances are of both the modeled values and actual values are listed in the table, and appear consistent upon visual comparison. In the case of all four metrics, the calculated Test Statistic is less than the t-distribution value. The null hypothesis may therefore be accepted, that there is 90% confidence that the mean of the modeled values is equivalent to the mean of the actual values for all four indicators of software product suitability.

Table 5. Model Results for Suitability Metrics

Quality Metric	Sample Size	Sample Size Needed	Modeled Value		Actual Value		Test Statistic	t-Dist Value	Means Equal?
			Mean	Variance	Mean	Variance			
Functional Adequacy	21	11	24.429	222.113	25.714	246.109	0.141	1.684	Yes
Functional Implementation Completeness	21	11	25.062	212.601	26.381	235.569	0.143	1.684	Yes
Functional Implementation Coverage	21	11	23.905	231.297	25.714	246.109	0.199	1.684	Yes
Functional Specification Volatility	21	20	2.105	23.460	5.667	92.032	1.299	1.684	Yes

6. Conclusions and Future Work

This research gives evidence that the proposed model can provide reliable predictions of software quality in terms of the metrics associated with the suitability of the software. While these results are encouraging, their validity will continue to be investigated through further data collection. In particular, it is hoped that more diverse software engineering projects, in terms of project scale and scope of quality, may be included. In addition, this research is planned to be expanded to include all of the software product quality sub-characteristics detailed in the ISO/IEC 9126 software product quality standard.

7. Acknowledgements

The authors would like to acknowledge the National Aeronautics and Space Administration at Kennedy Space Center for providing the resources to pursue this research.

8. References

[1] B. Kitchenham and S.L. Pfleeger, "Software Quality: The Elusive Target", *IEEE Software*, Volume 13, Issue 1, January 1996, pp. 12-21.
[2] N.E. Fenton and M. Neil, "A critique of software defect prediction models", *IEEE Transactions on Software Engineering*, Volume 25, Issue 5, September-October 1999, pp. 675-689.
[3] N. Fenton, P. Krause, and M. Neil, "Software Measurement: Uncertainty and Causal Modeling", IEEE Software, Volume 19, Issue 4, July-August 2002, pp. 116-122.
[4] S. Chulani, B. Boehm, and B. Steece, "Bayesian analysis of empirical software engineering cost models", IEEE Transactions on Software Engineering, Volume 25, Issue 4, July-August 1999, pp. 573-583.
[5] B. Cukic and D. Chakravarthy, "Bayesian framework for reliability assurance of a deployed safety critical system", Fifth IEEE International Symposium on High Assurance Systems Engineering, 15-17 November 2000, pp. 321-329.
[6] S.H. Kan, *Metrics and Models in Software Quality Engineering*, Addison-Wesley, Reading, MA, 1995.
[7] ISO/IEC, *Information Technology, Software Process Assessment, Part 1*, ISO/IEC Technical Report 15504-1, 1998.
[8] ISO/IEC, *Information Technology, Software Quality, Part 1*, ISO/IEC Standard 9126, 1995.
[9] V.R. Basili, L.C. Braind, and W.C. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators", IEEE Transactions on Software Engineering, Volume 22, Issue 10, October 1996, pp. 751-760.
[10] R. Subramanyam and M.S. Krishnan, "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects", IEEE Transactions on Software Engineering, Volume 29, Issue 4, April 2003, pp. 297-310.
[11] T.M. Khoshgaftaar, J.C. Munson, B.B. Bhattacharya, and G.D. Richardson, "Predictive Modeling Techniques of Software Quality from Software Measures", IEEE Transactions on Software Engineering, Volume 18, Issue 11, November 1992, pp. 979-987.