2006

# A Life Cycle Software Quality Model Using Bayesian Belief Networks

Justin Beaver
*University of Central Florida*

# A Life Cycle Software Quality Model Using Bayesian Belief Networks

by

Justin M. Beaver
B.S.E.E., Tennessee Technological University, 1995
M.S.Cp.E., University of Central Florida, 2001

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2006

Major Professor:
Guy A. Schiavone

# ABSTRACT

Software practitioners lack a consistent approach to assessing and predicting quality within their products. This research proposes a software quality model that accounts for the influences of development team skill/experience, process maturity, and problem complexity throughout the software engineering life cycle. The model is structured using Bayesian Belief Networks and, unlike previous efforts, uses widely-accepted software engineering standards and in-use industry techniques to quantify the indicators and measures of software quality. Data from 28 software engineering projects was acquired for this study, and was used for validation and comparison of the presented software quality models. Three Bayesian model structures are explored and the structure with the highest performance in terms of accuracy of fit and predictive validity is reported. In addition, the Bayesian Belief Networks are compared to both Least Squares Regression and Neural Networks in order to identify the technique is best suited to modeling software product quality.

The results indicate that Bayesian Belief Networks outperform both Least Squares Regression and Neural Networks in terms of producing modeled software quality variables that fit the distribution of actual software quality values, and in accurately forecasting 25 different indicators of software quality. Between the Bayesian model structures, the simplest structure, which relates software quality variables to their correlated causal factors, was found to be the most effective in modeling software quality. In addition, the results reveal that the collective skill and experience of the development team, over process maturity or problem complexity, has the most significant impact on the quality of software products.

*To the glory of God, who inspired this work, and carried it to completion.*

*And to my wife, Laura, for her unending encouragement and support.*

# ACKNOWLEDGMENTS

This research was made possible in part by funding from the National Aeronautics and Space Administration through the Kennedy Graduate Fellowship Program and the Kennedy Space Center Director's Discretionary Fund.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

## 1.1 Software Quality Overview

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. 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. However, high quality is a trade off - it is achieved with significant cost and effort. The challenge to the field of software engineering is to simultaneously maximize the quality of a software product while minimizing its associated cost and effort. Unfortunately, a lack of universal definition of software quality hinders the ability of a software development team to quantify software quality, and thus achieve the desired balance with cost.

Formulating consistent and reliable measurements to the quality of a software product is an enormous challenge itself. Even with adherence to a given definition or standard of software quality, the integration of objective and subjective measurement data makes analysis difficult. For example, how does one evaluate the usability of a system? There are certainly approaches to quantifying usability with software product and process measures, but it is difficult to ignore the role of the user opinion in such an evaluation. How can the subjective aspect of an evaluation be captured and presented along with the more objective measures?

Modeling software quality is multi-faceted: it involves both a current assessment of the software development effort, and a prediction of the quality of the delivered software product. A software development effort is a complex venture. It is comprised of many different factors that can affect the quality of the delivered system. Any model that attempts to capture the driving factors of software quality must be able to comprehensively represent all of those factors in order to provide meaningful insight. Such insight allows a development team to identify and address problem areas within an evolving software system, and the development infrastructure. Many of the prior software quality models are weak in terms of their ability to address the full spectrum of factors that affect software quality, their inability to be universally applicable, and their inherent data quality problems (discussed in detail in Sections 2.2.1.6 and 2.2.2.4). However, recent progress in software quality modeling, using complex adaptive systems, has shown promise in accurately representing causal relationships in software product development, and providing implementations that are adaptable to a given development team and environment.

## 1.2 Research Overview

This research effort addresses the need for a reliable approach to modeling the quality of software under development. The intent is to provide a model that will offer insight into the quality of the development products at a given stage in the life cycle, and provide dependable foresight as to the quality of the software product at the time of deployment. The ability to reliably assess and predict software quality gives a development team several advantages. From a technical perspective, a software quality model provides insight into the extent to which the software product meets its quality objectives, and reveals any quality objectives that are likely not to be met. This helps the development team manage the technical risks associated with software delivery to a customer, and insures that any probable technical weaknesses are adequately addressed prior to delivery of the software. From a cost and schedule perspective, software quality models are valuable in that they allow a development team to address forecasted problem areas earlier in th development life cycle, when they are least expensive and less time-critical to correct.

The software quality model developed in this research models cause-effect relationships throughout the development life cycle, and represents software quality as a function of three driving factors: the skill/experience of the development team, the maturity of the software development processes, and the complexity of the software product. This approach differs from previously developed software quality models in several ways.

1. No prior approach accounts for and relates the activities of the entire software development life cycle in modeling software quality. Typically, software quality models have focused on a specific life cycle phase (e.g., requirements, design) and relate measures from that phase to measures of software quality. The approach presented in this research accounts for skill, process, and complexity in all four life cycle phases (requirements, design, implementation, and integration/test) and models them as drivers of specific quality attributes.

2. This approach to software quality modeling is also unique in that it combines the effects of development team skill, software process maturity, and software problem complexity in assessing and predicting software quality. Previously proposed models focus on a specific set of measures, usually design complexity measures, and establish a relationship between those measures and an attribute of software quality. This research asserts that software quality is primarily affected by the three aforementioned causal factors. The value of a software quality attribute is determined by accounting for the influences of each of these three drivers.

3. Unlike previous software quality models, this research effort focuses on the use of existing standards and industry approaches to measuring the driving factors of software quality. Software product quality is characterized by the ISO/IEC 9126 Software Engineering Product Quality standard [ISO01]. The approach to measuring development team skill and experience is derived from a workforce management tool currently in use by the National Aeronautics and Space Administration [NAS05]. The assessment

of software process maturity is derived from the ISO/IEC 15504 standard [ISO98]. Software problem complexity is characterized by a combination of complexity measures established in the literature, and measures associated with the quality attributes identified in the ISO/IEC 9126 standard.

The unique elements of this approach to software quality modeling make this research an original contribution to the advancement of the field of software engineering.

The approach to modeling software quality in this research is to use Bayesian Belief Networks, a complex adaptive system, to represent the cause-effect relationships within the software development life cycle. Bayesian Belief Networks were selected because of their ability to explicitly model causal relationships, their ability to represent both objective and subjective data, their ability to adapt to prior data, and their performance in the presence of unknown or uncertain data. Three different Bayesian model structures are proposed and analyzed. Two of the structures models quality attributes as a function of correctness and completeness measures for each life cycle phase. One model structure captures a more direct relationship between model inputs and software quality outputs.

Validation of the proposed software quality model involved the comparison of forecasted values to actual values for each of the software product quality attributes represented. Two criteria were considered for comparison of the models: determination of Accuracy of Fit of modeled data to actual data, and a quantification of the Predictive Validity, or accuracy, of the model when forecasting. An analysis was performed between proposed Bayesian model structures, and versus competing software quality modeling methods.

## 1.3 Chapter Synopsis

The following paragraphs summarize the chapters that comprise this research effort to develop a software quality model.

In Chapter 2, a review of existing literature on software quality is presented. It begins with an overview of software quality including relevant definitions. A history of the approaches to modeling software quality is explored in detail, and a path of progress from simple correlations to complex adaptive systems is identified. This chapter also details previous approaches to the measurement of factors that influence software quality, such as problem complexity, development team skill, and process maturity. The literature review also explores existing approaches and standards for assessing software quality.

Chapter 3 describes the technical approach to developing a model for software quality. The scope of the research is presented, including a set of goals and objectives. The measurement frameworks selected for assessing both the drivers and outputs of the software quality model are detailed. The three proposed Bayesian Model structures are described in terms of the causal relationships between elements in the software life cycle. The approach to validation of the model is also presented in this chapter, including the set of criteria used to verify the model's ability to assess and predict for the various software product quality measures.

In Chapter 4, the results of the various analyses are presented. Initially, the model variables are scrutinized in terms of their variability and their multicollinearity with other variables. An analysis of each Bayesian Model structure is performed, as well as an assessment of the impact of the various causal factors of software quality. The Bayesian model structures are compared to each other, and then Bayesian Modeling is compared to both Least Squares Regression, and Neural Networks. The results indicate that the Bayesian approach models software quality more accurately in terms of the Accuracy of Fit and Predictive Validity criteria.

Chapter 5, the conclusion, provides a commentary on the results of the research. The viability of Bayesian Belief Networks as a software quality modeling tool is discussed as well as a summation of the performance of the measurement frameworks employed. This chapter also touches on those factors that were found to have the greatest influence on the quality of a software product. It concludes with a discussion of the practical application of the developed software quality model in an industry setting.

# CHAPTER 2

# LITERATURE REVIEW

This section outlines the previous research that defines, evaluates, and models software quality. An overview is provided of software quality in terms of definitions and interpretations. The evolution of software quality modeling is then presented, beginning with correlations used to predict software quality and progressing in maturity to the most recent use of complex adaptive systems. Finally, the measurement frameworks in the literature used to represent the various factors surrounding a software development effort are discussed.

## 2.1 Defining Software Quality

Software quality is a term that is intended to capture the excellence of both the functional and non-functional aspects of a delivered software product. Its use has ranged in scope from describing how well the software product meets the requirements to describing how portable the software is between platforms to describing how aesthetically pleasing a given software product's interface is. In short, software quality is used to describe the effectiveness, efficiency, flexibility, reliability, robustness, and usability of a given software product. Software product quality cannot be solely objectively measured, and therefore has endured much debate as to its true definition. Perhaps Dromey [Dro96] characterized the nature of software quality best by saying,

> ”It helps to get clear at the outset that some very elusive notions - like ’quality,’ ’goodness,’ and ’fitness-for-purpose’ - are experiential. That is, people make a judgment, depending on their particular needs or perspective, that something they use, encounter, or examine is ’good’ or has ’quality.’ Exactly what tangible properties engender such a response is something quite different. In our quest to improve software quality, we must devote much more attention to this area.”

The passage implies that the burden lies on the software engineering community to establish relationships between measurements and the subjective assessment, and to create an approach to modeling that takes as its inputs both subjective and objective data.

In order to understand, assess, and model software quality, it is appropriate to at least put forth a definition of software quality. There is no shortage of definitions proposed that characterize the quality inherent in a software product. In many cases, definitions differ significantly based on the perspective of the assessor. Kitchenham and Pfleeger [KP96], for example, identify five different views of software quality that have very different definitions and goals. They point out that software quality for the end user may fall along the lines of fitness of purpose, while quality from a manufacturing perspective may focus more on the adherence of the software to its requirements. For the purposes of this research, the definition of software quality from the Handbook of Software Quality Assurance [SM99] shall be modified to define software quality as follows:

”Software quality is the fitness for intended use of the software product.”

This definition has many implications. It encompasses the measurable elements of a software product, such as conformance to functional requirements, and the less measurable elements of a software product, such as ease of use. The ambiguity associated with the definition is appropriate given the goal of incorporating both subjective and objective information into a software quality assessment and prediction model. In addition, this definition of software quality is a very customer-centered definition. The fitness for use of a product is ultimately determined by the set of people that will be the users of the software. A software quality definition that implies a reliance on user acceptance increases the likelihood that a high quality software product will invoke customer acceptance.

## 2.2 Evolution of Software Quality Modeling

The modeling of software quality is a risk management approach that serves two purposes: to assess the current state of the software development effort, and to predict the final state of the software development effort. This section provides a synopsis of the evolution of software quality modeling during the development life cycle. Initially, software quality models were simply empirical correlations established between a particular metric and a particular element of software quality. As the complexity of the software development process became evident, software measurement research turned to intricate multivariate statistical models to more accurately represent a software product. The statistical models proved inadequate for global application, which has bent software quality models to explore more adaptive algorithms to more accurately model the quality of a developing software product.

### *2.2.1 Modeling Software Quality: Correlations*

Early work in using software engineering metrics to characterize software quality focused on simple empirical correlations between the various metrics and the likelihood for product faults or changes. These studies would typically present a particular design/implementation metric as a new way to measure a software product, and then validate that metric as a quality indicator by correlating it to some software quality attribute (typically the number of faults discovered before operational use, or the number of changes to a particular software module). This section presents several of those early papers and their contributions to software quality modeling. Section 2.2.1.6 then presents the challenges and critiques to these early approaches to software quality modeling.

#### *2.2.1.1 Cyclomatic Complexity*

McCabe's Cyclomatic Complexity metric [McC76] is perhaps the most well known instance of establishing a software engineering metric. Graph theory is used to compute the minimum number of paths through the program control structure, and calculate the metric. The value of the minimum number of paths (a positive integer) is said to represent the complexity of the program's control flow, also called the Cyclomatic Complexity. The measure was not validated in this paper but a general rule of thumb was proposed that a Cyclomatic Complexity of 10 is the complexity threshold. That is, if the calculated Cyclomatic Complexity exceeds 10, it is deemed too complex, and it is recommended to further modularize a given component. In addition to establishing a mathematical foundation for quantifying program control flow, this measurement approach was revolutionary because its method of calculation is dependent on program structure and independent of a specific programming language. In addition, a value of program complexity can be derived at the time of design that remains constant throughout implementation and test, assuming that the structure of the implementation represents the design.

#### *2.2.1.2 Software Science Metrics*

Halstead's Software Science metrics [Hal77] are based on the grammatical structure of the source code and are used to characterize software size, and estimate programming effort.

In Software Science metrics, the grammar of a program is classified as either an operator or an operand, and size calculations are made based on the variety and quantity of each. Smith [Smi80] corroborated the accuracy of Halstead's approach to assessing and predicting the software size with industry data. Although Halstead's metrics were not intended to be a predictor of the number of faults or changes, several studies have been performed to make just that connection [EKC98] [Nai82] [KC85]. The results have been varied which calls into question the universal applicability of these metrics as quality indicators. Another weakness in these measures is that they depend heavily on the completed code, which limits their effectiveness as predictors of software quality. That is, the code must be complete before the measurements can accurately depict the quality aspects of the software under development. Since the completion of code is relatively close to the time at which faults are discovered, predicting a fault distribution does not provide much value.

#### *2.2.1.3 Information Flow*

Henry and Kafura [HK81] performed an early study that established information flow metrics as measures of interface complexity, and related those metrics to the number of changes in a software module. Follow-on studies [HK84] [KC85] performed by the same authors further validated the effect of interface complexity. The conclusion reached by this research effort was that the squared product of two of the information flow measures, Fan-in and Fan-out, were highly correlated to the number of changes in a given software module. The study provided no threshold for information flow metrics as an indicator of high complexity, but rather took a more relative approach to identifying problem modules within the development effort. That is, it concluded that those modules within the project that had the highest information flow value were also the most likely to have a largest number of code changes. The use of information flow was not as successful in other studies [KPL90] [She90]. The alternative results are discussed in detail in Section 2.2.1.6.

#### *2.2.1.4 Lines of Code*

The Lines of Code (LOC) metric is perhaps the most well known measure of software size. LOC is used regularly in government and industry as the basis for assessing software project cost estimates, worker productivity, staffing needs, etc. Despite its lack of established definition and its unreliability in establishing an estimate, the use of LOC in estimating and predicting aspects of software development is broad.

The use of LOC in empirical studies is equally as broad. In some empirical studies, LOC is a factor used in the prediction or assessment of some aspect of software quality. LOC is validated as having an inverse relationship with the defect density [Wit90] of a program. LOC is been used to predict the reusability of a software component [TS91]. LOC is also often used as the size parameter in assessing or predicting project management aspects of the software effort: project cost [BBL76], effort [GBB90] [MK92], or size [Dol00] [MK92].

#### *2.2.1.5 Chidamber and Kemerer*

In 1994, Chidamber and Kemerer [CK94] published a set of six metrics, known as C&K metrics, which were specifically aimed at evaluating the design of an object-oriented (OO) system. The intent of the metrics suite was to serve as a tool for assessing the quality of the software being developed, and to provide managers with data to provide insight into the quality of the design. Li and Henry [LH93] analyzed the relationship between the C&K metrics and the effort required in program maintenance. Using the data of two industry systems, five of the six C&K metrics were correlated to the amount of code change. Basili, et al, [BBM96] used data from several small and low complexity student projects to empirically validate five of the six Chidamber and Kemerer metrics as a predictor of fault-proneness in a given class. More recently, Subramanyam and Krishnan [SK03] empirically validated a subset of the C&K metrics using industry data. This study, unlike its predecessors, provided empirical evidence after taking into account the confounding effect of the size of each class.

#### *2.2.1.6 Challenges to the Validity of the Empirical Correlation Approach*

As more metrics became available and the claims of correlation to aspects of software quality became more common, the validity of the empirically established relationships began to come into question. Researchers who had less than optimal success in applying these metrics as software quality predictors began to report their findings.

Khoshgoftaar and Munson [KM90] measured the level of association of several metrics, including the Lines of Code metric and different control flow metrics, to the number of program errors. Using four different regression models, these metrics were tested for their statistical relationship to the number of program errors. The researchers found that none of the four models exhibited a relationship between Lines of Code and program errors.

Kitchenham, Pickard, and Linkman [KPL90] performed a study in which the Henry and Kafura Information Flow metrics were attempted to be validated in a 226 program system. The result was not as favorable as the Henry and Kafura study. The information flow metric was found to be only weakly correlated with the number of program changes or the number of program errors.

Sheppard's [She90] research attempted to discern a correlation between the Henry and Kafura information flow metrics and the development time of a project. His results found no correlation using the original information flow metric, but a high correlation using a modified form of the Henry and Kafura measure. The researcher proposed that the original metric relies too heavily on indirect flows in its calculation of the information flow, which skews the result. The research stated that the version of the information flow metric that was calculated using unique flows was a more accurate predictor of project development time.

In 1994, Briand, Morasca, and Basili [BMB94] attempted to identify error-prone modules in software development projects using high-level design metrics as indicators. Over 1000 Ada modules were analyzed, and different modeling techniques and metrics were compared to determine their ability to identify error-prone modules. Among their findings the researchers discovered that the information flow metric was not statistically significant.

El Emam, et al. [EBG01], brought into question the validity much of the empirical research on object-oriented metrics by identifying the confounding effect of class size. In the study, the Chidamber and Kemerer object-oriented metrics were applied to a telecommunications framework, and analyzed for an empirical relationship to the fault-proneness associated with each class. As in previous OO metrics studies, a correlation was established. However, when the same data was controlled for class size, none of the metrics retained the correlation. The authors state that the confounding effect of class size threatens the validity of prior OO metric research, and that future empirical studies of OO metrics should be controlled for class size.

A study performed by Lanubile, Lonigro, and Visaggio [LLV95] compared the predictive validity of seven different statistical models using eleven different complexity metrics. The method of comparison was a risk assessment of each of 118 modules using the chi-square statistic as a determination of statistical significance in predictive validity. Interestingly, the paper found no statistical significance with respect to either the metric set used, or the model used. The authors attributed this lack of significance to the fact that results are specific to environments, and that no generalization may be made.

Capers Jones identified in a brief IEEE Computer article [Jon94] the most effective and ineffective metrics used in software engineering. The article argues that the Lines of Code metric and Halstead's software science metrics are ineffective when used to compare productivity information since they are so programming language dependent. The cost-per-defect metric is argued to be unproductive since it easily misrepresented (more efficient code is penalized by this metric). Jones identifies the complexity metrics and function points as software engineering metrics that do work. He argues that these metrics are valid as indicators of the quality of the software development effort due to their independence from implementation and their availability in the design phase.

The early studies on software engineering metrics as quality indicators focus more on establishing rules of thumb than applying a statistical model to the data. That is, these research contributions seek more to establish acceptable thresholds for their derived metric than to model statistically the relationships between software metrics and various indicators of product quality. The varying results according to the data sets used were indicative of the very specific application domain for many of these measures. That is, the validity of the information flow metric as a predictor of the number of changes appears, from the varying results, to be highly dependent on the project-specific data.

Courtney and Gustafson [CG93] highly criticized the use of empirical correlations in software engineering research to establish statistical relationships. They argue that small sample sizes and attempting to correlate several independent variables to a single dependent variable is not sound statistics, but rather a "shotgun" approach to statistical significance. In their own words, "Owing to the large number of interacting factors and the limited amount of data, the likelihood of finding accidental relationships is high." To illustrate their point, the authors performed an experiment in which a random number generator was used to produce measures in the range of Halstead's software science metrics. Using the "shotgun" approach, they were successful in establishing a correlation between the random independent variable, and a random dependent variable. Courtney and Gustafson also state that the lack of a large body of careful and consistent software engineering measures has stunted the development of software engineering statistics because conclusions are specific to an environment.

The weakness of the early approaches to predicting the quality of software was the assumption that a single measure was the panacea as an indicator of the quality of a given software module. As researchers became more aware of the complexity of software development and of the quantity of possible indicators of software quality, it became apparent that the software development effort was not sufficiently characterized by simple treatment-effect correlations. The field of empirical software engineering began to progress toward more complex statistical approaches to capture, represent, and predict a software development effort.

### *2.2.2 Modeling Software Quality: Multivariate Models*

The next step in the progression of software modeling techniques was to represent and determine particular attributes of software quality through the use of multivariate statistical models. The primary focus of these models was to identify high-risk software modules using metrics derived early in the development life cycle. These models sought to more accurately capture the complexities of the software development effort by representing several different independent variables and the relationships between those variables to arrive at an indicator for fault-proneness of a given module.

The use of multivariate models was an important step in software quality modeling because it built on the empirical validation of previous research, and incorporated a set of those validated metrics into a statistical model. Basili, Briand, and Hetmanski [CC00a] identified this process as a two-step approach to developing a multivariate model:

1. Define and validate metrics that are good predictors of fault-proneness, and
2. Select an appropriate modeling technique, in terms of underlying assumptions, to yield the most accurate prediction.

Where empirical validation simply validates individual metrics as indicators of software quality, the use of statistical methods combines those metrics into a statistically valid approach to predicting the quality of a given software component. The subsections below identify some prior research in the use of statistical models to model software quality, and ends with identifying the weaknesses associated with the approach.

#### *2.2.2.1 Analysis of Variance*

Zage and Zage [ZZ93] proposed a design metric to identify the fault-prone aspect of software quality for modules in a software development effort. The design metric consolidated the external and internal components of design complexity into a single metric. The approach to the identification of fault-prone modules was to calculate the average design metric value for the modules on a project, and determine the set of modules that exceeded a single standard deviation from the average. These were the modules that were highlighted as the most fault-prone. The design metric and the approach to fault-prone identification were validated on 21 Ada programs from a Department of Defense project, and using McCabe's cyclomatic complexity and Lines of Code as control groups. The researchers found that their proposed metric was more accurate than the two other approaches in identifying fault-prone modules.

Bandi et al. [BVT03] conducted an empirical study in which three measures of object-oriented design were evaluated for their effectiveness in predicting the maintenance performance of software under development. Each of the three design metrics (Interaction Level, Interface Size, and Operation Argument Complexity) was found in an Analysis of Variance to be a significant impact to the amount of maintenance time required for a given module. In addition to correlating these complexity metrics to maintenance time, the study determined through a regression analysis that there is sound evidence of multicollinearity between these three metrics. Thus, in a practical sense, analysis of only one of the three metrics would suffice in projecting maintenance effort.

#### *2.2.2.2 Regression Models*

Regression is a statistical approach in which the mean value of a dependent variable is expressed as a function of one or more independent variables. In linear regression, the relationship between dependent and independent variables is expressed in the form of a linear equation. In logistic regression, an equation based on maximum likelihood estimation equation is used to relate the dependent and independent variables. A multivariate logistic regression model expresses the dependent variable in terms of the set of independent variables. Univariate logistic regression is a special case where only one independent variable is used. Regression models were used extensively in empirical software engineering literature as a means of establishing apparent statistical relationships between a set of design characteristics, and a software quality measure.

Briand et al [BDP98] used logistic regression in 1998 to explore the relationship between several object-oriented design metrics and the probability of fault detection in a given class. The data used in this research effort was a set of student projects from which design measures were derived after implementation. For each collected measure, a suitable distribution and significance in terms of predicting fault-proneness were required to be included in the multivariate analysis. Of the sets of coupling, cohesion, and inheritance measures used as independent variables, the class size, frequency of method invocation, and depth of inheritance measures were the most strongly related to class fault-proneness. The authors did not appear to address the issue of interaction between independent variables. That is, it is unclear whether any considered independent variable had a confounding effect on the significance of another independent variable.

Khoshgoftaar, et al [KMB92] performed a study in 1992 in which four different approaches to regression analysis were compared to determine the most effective at predicting the number of changes in a given software module. Applying the four models to two separate data sets and using the average relative error as the measure for accuracy, the researchers determined that the Relative Least Squares and Minimum Relative Error estimation techniques were the most accurate in terms of both quality of fit and predictive accuracy.

Khoshgoftaar and Seliya [KS02b] performed a case study that compared different regression tree models for use in predicting the quality of high-assurance software. The subject system was a large telecommunications system with approximately 13 million lines of code. Design metrics (call graph and control flow graph measures) were used as inputs to the regression trees that were compared for prediction accuracy and complexity. The average relative error and average absolute error were the calculations performed to assess accuracy. The study discovered that one of the three models used, the CART-LAD tree, was effective in terms of its accuracy and complexity. This study was performed a posteriori, and no mention was made as to the rigor included in the design and development process. That is, it was unclear from the study whether the rigor of the design process was a factor in the accuracy of the model or the ability of the design measures to represent the underlying system. This study mirrors many studies involving design metrics as quality indicators: it validates a specific model in a specific situation, but does not comment on the applicability of that model on other software projects, nor does it comment on the effect of rigor in the development process on the quality of the developed software.

#### *2.2.2.3 Other Multivariate Models*

Beaver and Linton [BL02] proposed a method for identifying error-prone design modules in a system by analyzing four different Information Flow measures. The technique used spatial data analysis to create a surface based on Information Flow metric values in a training data set, and modeled the extent to which a module was likely to be error-prone. The technique was found to be a good predictor for design-module error-proneness. In a follow-on study, Beaver and Schiavone [BS03a] found the spatial data analysis technique to be superior to both Least Squares and Relative Least Squares Regression techniques in terms of its ability to predict error-prone design modules.

#### *2.2.2.4 Challenges to the Validity of the Multivariate Model Approach*

In 1999, Fenton and Neil [FN99] performed an extensive critical review of existing techniques for predicting software quality. Several of the papers cited in this literature review [Hal77] [McC76] [CK94] [BBM96] [KMB92] were addressed in the Fenton and Neil critique. The review outlined several problems with existing approaches including the following:

1. Unknown Relationship between Defects and Failures

   The authors identify that inconsistent terminology exists for defects/failures. Thus, it is difficult to determine exactly which definition each presented model claims to predict. In addition, they propose that defect counts are misleading as a measure of software reliability due to most studies' neglect of classifying defects to determine severity.

2. Problems with Using Size and Complexity Metrics Exclusively

   This paper states that models that are built on size and complexity metrics assume a direct and exclusive relationship between those measures and the faults discovered. These models fail to take into account factors such as programmer/designer ability, and problem difficulty.

3. Weak Statistical Methodology/Data Quality

   The authors indicate that several statistical assumptions are ignored in the development of software quality prediction models. In the case of linear regression models, correlation between predicting variables is assumed to be zero, and often is not. It is also stated that regression approaches are typically centered on fitting models to data rather than predicting data. That is, a regression model may accurately depict historical data, but is not necessarily successful at predicting. Furthermore, the paper emphasizes that removing data and using averaged data is a common practice in these studies and undermines the integrity of the results.

The authors assert that these inherent statistical problems seriously threaten the validity of the collective findings. A study by Henderson-Sellers [Hen96] echoes the points raised by Fenton and Neil, and argues that, *"the misapplication of quantitative techniques together with errors in the underlying mathematics"* have undermined the credibility of the software engineering community's approach to metrics and measurement.

In a follow-on study, Fenton et al. [FKN02] reinforce their position by discussing the dangers of relying exclusively on size/complexity measures as indicators of software quality, and emphasize the importance of creating prediction models which represent cause-effect relationships in a system. The study recommends Bayesian Belief Networks as an approach that is better suited to the complex conditions surrounding software development. This paper represents a turning point in the modeling of software quality. They identify the insufficiencies of representing software quality with design measures exclusively, and without regard to the myriad of factors that are causes of software quality. The Fenton papers propose that Bayesian Belief Networks more appropriately address the problem of modeling software development because they are well-suited to both integrating objective measures with subjective influences, and making predictions with little or incomplete data.

### *2.2.3 Modeling Software Quality: Complex Adaptive Systems*

The most recent step in the evolution of software quality modeling techniques was the application of complex adaptive systems. These approaches provide unique benefits over traditional statistics in modeling the complexities of software development. Complex adaptive systems are designed to predict. They are models that represent logical chains of decisions, algorithms, or cause-effect relationships that have been weighted using prior data sets, and forecast the most likely output for a given set of inputs. The use of prior data to "train" the model is a distinct advantage in that it allows the models to be customized to an organization's local data. In addition, complex adaptive systems perform well in the presence of incomplete/uncertain data and can represent both objective and subjective data types. This section outlines several different complex adaptive systems that have been employed as methods for creating predictive models of software quality.

#### *2.2.3.1 Decision Trees*

A decision tree is a directed acyclic graph in which the outputs from each node represent a decision made based on a single input situation. The structure of a decision tree outlines a logical sequence of decisions that need to be made to arrive at the goal decision (the question the decision tree attempts to answer). Analyzing a training set of data allows for the determination of the decision tree structure. Decision nodes that isolate the goal decision quickly are prioritized to the top of the tree, while more evenly distributed decision outputs are reserved for the bottom of the tree. Decision trees have been demonstrated in software engineering empirical research [MDC03] [KS02a] [SP88] [KAB96] as a support tool in assessing and predicting various aspects of software quality.

#### *2.2.3.2 Neural Networks*

A neural network is a topology of nodes connected by links. Each node in a neural network has a set of input links coming from other nodes, and a set of output links driving to other nodes. A node's activation function is the characteristic of the node that transforms the weighted sum of input values to an output value. Each link in the network has a weight associated with it that designates its influence over the subsequent node. The weights associated with each link may be adjusted in value based on a given set of data to bias the network to certain patterns of output. A neural network is usually represented graphically, but can also be represented mathematically given that the activation function can be represented mathematically.

Khoshgoftaar has authored several papers using neural networks as a tool for predicting software quality, including [KSG95], [KAH97], and [KS96]. In these papers, neural networks were used to classify software modules as fault-prone based on a set of design complexity indicator metrics. Principal Components analysis was used to identify the relationships between the various indicator metrics and the propensity for faults. The results when using a neural network to model the effects of the principal components revealed that the neural network was a good predictor of module faults, and performed well in terms of its ability to correctly classify modules.

Quah et al [QT03] used neural networks to predict the quality of a software product in terms of reliability and maintainability. Reliability was measured as the number of defects in a given software object, and maintainability was quantified as the number of changes to a given object. The research used object-oriented complexity measures as the independent variables and used both the Ward and General Regression neural networks to compare predictions. The study determined that while both neural network techniques performed well, the general regression neural network was a better predictor for both software quality variables.

Kanmani et al [KUS04] applied a general regression neural network to predict software quality. The study used 64 object-oriented measures as the independent variables and attempted to predict the value of the fault ratio, a measure of the proportion of classes that were found to contain a fault. The results of this study indicated that the general regression neural network adequately represented the underlying fault ratio data, and was a good predictor of the fault ration metric.

#### *2.2.3.3 Genetic Algorithms*

Genetic algorithms are an approach to machine learning in which a data point is selected from a population based on defined fitness and reproduction functions. The approach is intended to emulate Darwin's theory of natural selection (an interpretation of the evolution of species in nature). The premise behind genetic algorithms is that by correctly describing the fitness and reproduction functions, a response to a given set of inputs will be selected from the data set that represents the most likely response for that input criteria.

Evett et al [EKC98] elected to use a genetic programming algorithm in order to make predictions about the reliability of software modules under development. The goal of this research is to identify, in rank order, those software modules that are most likely to encounter faults such that those can be targeted for reliability enhancement. Using two industry data sets, and splitting each set into a training and validation subset, the genetic algorithm described in the paper was applied to determine the rank order of modules most likely to encounter faults. The inputs to the model were a collection of software product metrics that can be captured early in the development life cycle (Halstead's, Cyclomatic Complexity, Lines of Code), and the output was an ordered list of fault-prone modules. The research found that the top 10% of the outputted lists accounted for 79% to 86% of the known faults in the industry data sets. The authors expressed that their approach was specifically suited to identifying modules that are eligible for reliability enhancement, and could be valid as an indicator of software quality.

Azar et al [APB02] used a genetic algorithm to determine the most appropriate software quality prediction model to apply on a software project. In addition to the model selection, the research proposed ways to combine and/or adapt models in a way that best suited a given development organization. The results were mixed. In one case, the output produced a classification tree which combined the knowledge of many experts, and produced a model that performed well in predicting software quality. In another case, the genetic algorithm produced a model, based largely on the knowledge of a single expert, that did not perform as well.

#### *2.2.3.4 Bayesian Belief Networks*

A Bayesian Belief network is a directed acyclic graph (DAG) in which each node represents a random variable, and each arc represents a causal relationship between the joined nodes. The graph is called a Bayesian Belief network because the random variables that each node represents are distributed according to the conditional probability of that node based on its inputs. The determination of probabilities associated with a node's outcome is based on Bayes' Rule, a conditional probability equation. When applied to the random variables within each node, Bayes' rule becomes a powerful tool in limiting the state space for calculating conditional probabilities over a network of nodes.

Research by Cukic and Chakravarthy [CC00b] demonstrates the applicability of the Bayesian framework to a safety-critical system. A Bayesian approach was taken to evaluate the reliability of a deployed system that controlled spacecraft launch guidance commands. The Bayesian approach was chosen because its framework allowed for the incorporation of previous program executions as a model input. Until this framework was used, that data point was not available as an input into the reliability model. The calculations of reliability for this mission-critical system revealed that the system fell within tolerances for failure rates.

Chulani [Chu01] elected to use the Bayesian approach in creating a model that predicts the cost, schedule, and quality of a software product under development. The advantages of the Bayesian network cited by the author are its ability to cope with scarce data, and its capacity to factor both qualitative and quantitative data into the analysis. In addition, this study compared the Bayesian approach with the more traditional regression analysis approach for accuracy of predictions over 161 software projects. The Bayesian model was found to be significantly more accurate in predicting cost, schedule and quality factors.

Smidts, Sova, and Mandela [SSM97] employed a Bayesian framework in order to quantify the reliability of a software effort by deriving its failure modes. The Bayesian approach was chosen for this problem because 1) data is rare for software development efforts, 2) software failure data is comprised of subjective and objective considerations, 3) data may be weighted according to relevance, and 4) data points may be derived from engineering analysis and experience. The model was constructed using the functional architecture as a basis, and augmented with nonfunctional elements such as quality factors, external interface requirements, economic requirements, etc. When applied to a simple example, the authors found their model to be a bit too conservative, but nonetheless felt it was applicable and scalable to other software development projects. This study's contribution is its identification and implementation of the Bayesian approach as the best method for modeling software under development. This study was specifically aimed at quantifying the reliability of the developed software. Although their findings are preliminary, they hold promise for the Bayesian approach as a valid technique for modeling software under development.

Another application of the Bayesian approach was in the assessment of the dependability of a safety-critical system by Fenton et al [FLN98]. The authors sought to more accurately quantify the measure of dependability, which traditionally is a subjective assessment, by combining a wide variety of evidence such as failure data, engineering judgment, and team competency. Bayesian Belief Networks were chosen as the model to assess dependability because they provide a rigorous technique that can account for future uncertainties based on the objective and subjective evidence available. Once the Bayesian model was built, the researchers exercised it to invalidate two common assumptions in the development of safety-critical systems: 1) that experience developing 'similar' products does not increase the dependability of the system, and 2) that independent redundant systems improves the dependability of the system. This paper concluded that although the predictions made by the model cannot be validated without more complete data, the benefit of this approach is that the assessment of dependability is now quantifiable and repeatable.

In a recent IEEE Software article [FKN02], Fenton, Krause and Neil make a strong case for Bayesian Belief Networks as a predictive model that can provide advance warning of potential risks in software development. They support their case by pointing out the ability of BBNs to represent "genuine cause-effect relationships". That is, they hypothesize that complexity measures, process maturity, and fault data only make up a part of the causal relationship with software quality. The rest of that relationship is comprised of more subjective measures such as team capability and test effectiveness. It is the Bayesian approach that can accurately integrate the diverse set of data required to create a correct causal model. The authors claim that the construction and execution of BBN models will provide reliable data for projects to make more informed risk management decisions.

## 2.3 Causal Factors of Software Quality

What causes software quality in a product? The bulk of software quality literature to date deals with identifying associations between software development measures and software quality measures. Trends among various measurements are identified, but the issue of causality is insufficiently addressed. 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. For example, if a study identifies an empirical relationship between the number of lines of code in a software component and the corresponding number of defects, is it correct to assume that large programs cause more defects? Would a viable programming solution be to consolidate several lines into a single statement to reduce program size? Such a study may have executed a perfectly correct statistical experiment, and yet has provided little value to the software engineering community.

What establishes a cause-effect relationship, and how can it be scientifically validated? Stephen Kan [Kan95] has cited the following three criteria for using empirical data to establish a causal relationship:

1. In a causal relationship between two variables, the cause precedes the effect in time or logic.

2. In a causal relationship between two variables, the two variables must be empirically correlated.

3. In a causal relationship between two variables, any observed correlation is logical.

Most prior research in software quality modeling has sought to satisfy conditions 1 and 2 of the causal relationship, and yet has ignored condition 3. Thus, the logical component of establishing a cause-effect relationship is not addressed.

The following subsections explore the existing literature that addresses the three causal factors of software quality proposed by this research. These three factors attempt to satisfy the logical component of Stephen Kan's criteria. The intent is to demonstrate an existing body of research that supports these three factors as causes of software product quality.

### *2.3.1 Factors in Software Product Quality: Software Process*

Both industry and academia have provided strong evidence that an improvement in an organization's processes for software development can be associated with an improvement in software product quality. Capers Jones [Jon96] presented the costs and associated benefits of software process on the quality of software. He identifies a reduction in defect density, an increase in software productivity, and an increase in the volume of reusable software as the benchmarks for the quality improvement. El-Emam and Briand [EB99] performed an analysis of the costs and benefits of several software process improvement (SPI) initiatives. In their results were listed several industry examples of significant cost, schedule, and quality improvements that resulted from the associated SPI efforts. In addition to those findings, there is plenty of industry testimony [DS97] [Hal96] as to the improvements to the quality of the software product witnessed by an organization when SPI was implemented.

The implementation of SPI in an organization is typically accomplished by following an SPI model. The SPI models categorize similar processes into groups that collectively address the optimum activities in the life cycle of a development effort. Within each process category the SPI models identify practices that a development organization must perform to demonstrate their effectiveness and efficiency. This effectiveness/efficiency is referred to as the organization's maturity or capability. The premise is that a defined, consistent, and methodical approach to engineering software problems maximizes an organization's ability to be successful. The most frequently used models for software process improvement are detailed in Table 2.1.

Table 2.1: Widely Used Software Process Improvement Models

| SPI Model | Managing Organization | Focus | Validation Studies |
|---|---|---|---|
| Capability Maturity Model Integration, Version 1.1 [HS87] [Ins02] | Software Engineering Institute,<br>Carnegie Mellon University | Software Engineering<br>Systems Engineering<br>Project Management | [DS97] [Hal96]<br>[HG96] |
| BOOTSTRAP | BOOTSTRAP Institute | Software Development | [ESL97] [HMK94] |
| ISO 9000<br>ISO 9000-3 | International Organization for Standardization (ISO) | Quality Assurance<br>Software Development | [HJP98] [Ben95] |

This research effort is concerned with the effect of process on the quality in the software product. As such, the quantitative assessment of organizational process capability is critical to mathematically representing process maturity. Although each model is accompanied by a quantitative assessment method, a common framework for assessing process capability is more desirable in terms of the applicability of the research.

The International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) has established a framework for software process assessment and improvement called ISO/IEC 15504 Information Technology - Software Process Assessment [ISO98]. The ISO/IEC 15504 is an international collaboration, managed by the Software Standards Working Group in the ISO/IEC Information Technology Task Force (ISO/IEC JTC1/SC7/WG10) that creates a global standard for process assessment, process improvement, and organizational capability determination. It is important to note that while the ISO/IEC 15504 is not a model for software process improvement, it does provide a common framework for assessment of an organization's capability.

The ISO/IEC standard provides a two-dimensional view of software development processes: a set of process categories and a six-level rating associated with each practice in those categories. The five process categories are listed in Table 2.2.

These five process categories consist of 40 specific processes that represent activities that need to be performed in a mature software development organization. In addition there are 9 common processes that apply across all process categories. Assessing the maturity of an organization is done by way of rating each process as one of six capability levels depicted in Table 2.3. In addition to establishing the capability levels, the ISO/IEC 15504 provides guidance for assessing software organizations. The standard is intended to be a common approach to software development organization capability determination.

Table 2.2: Descriptions of ISO/IEC 15504 Process Categories [ISO98]

| Process Category | Description |
|---|---|
| Customer-Supplier | Process that impact the customer, support the development and transition of the software to the customer, and provide for the correct operation and use of the software product and/or service. |
| Engineering | Processes that directly specify, implement, or maintain the software product, its relation to the system, and its customer documentation. |
| Support | Processes that may be employed by any of the other processes at various points in the software life cycle. |
| Management | Processes that contain generic practices that may be used by anyone who manages any type of project or process within a software life cycle. |
| Organization | Processes that establish the business goals of the organization and develop process, product, and resource assets that, when used by projects in the organization, will help the organization achieve its business goals. |

Table 2.3: ISO/IEC 15504 Capability Levels [ISO98]

| Level | Name | Description |
|---|---|---|
| 0 | Incomplete | Process is not performed. |
| 1 | Performed | Process is performed, but generally not planned or tracked. |
| 2 | Managed | Process is performed, planned, and tracked. Process incorporates elements of quality assurance for work products. |
| 3 | Established | Process is based on established software engineering best practices and defined for the developing organization. |
| 4 | Predictable | Process is performed consistently and with defined control limits. Detailed measures of process performance are collected and analyzed. |
| 5 | Optimizing | Continual process improvement is enacted using quantitative process performance measures. |

### *2.3.2 Factors in Software Product Quality: Problem Scope*

Most of the software engineering research to date addresses the effect of elements of the problem scope on the quality of a software product. Problem scope is a term that is meant to include any attribute of a software solution that represents the size, complexity, or variety of applications of a software development effort. The typical approach in assessing software quality by problem scope is to characterize and quantify the attributes of a software development effort, and validate that quantification by establishing statistical significance with a specific software quality metric. In spite of the fact that much of the prior research has been riddled with both data quality and experimental design problems (see Section 2.2.2.4), the causal relationship between problem scope factors and software quality attributes is both logically and empirically evident. The discussion in Section 2.2 cites numerous examples of relating an attribute of the software problem to an attribute of software quality.

### *2.3.3 Factors in Software Product Quality: Personnel Skill/Experience*

Perhaps the least amount of existing research deals with the effect of the personnel involved in a software development effort on the ultimate software product quality. James Bach has authored several articles [Bac99] [Bac95] insisting that the quality of the people is the primary driver for software quality, and that too much industry focus has been on the process. He argues [Bac99] that personal performance is *"guided by higher level process models embedded within experience, education, and insight."* Certainly, from both an intuitive and experiential perspective, the talents and abilities of an individual play an important part in their success as an engineer, and the ultimate quality of their developed software. However, there is little research that supports the logical assumption.

Puerta and Carnal [PC89] performed an experiment with students in which the goal was to validate the fitness of Halstead's Software Science Metrics as factors in a linear software quality model derived by the authors. Although a subset of the Halstead measures were correlated with the set of software quality indexes, a measure of "programmer rating", based on the student's GPA was found to be the most highly correlated with high quality indexes. The authors merely reported the correlation, and did not comment on the implications of their finding.

The Competency Management System (CMS) [NAS05], developed by the National Aeronautics and Space Administration (NASA), is an existing approach approach to assessing the skill and experience associated with a software developers. The CMS is an enterprise application used to quantify the knowledge, skills, and abilities of the NASA workforce, and to intelligently manage the allocation of that workforce those projects where appropriate needs exist. For each technical discipline (e.g., Software Engineering, Systems Engineering, Chemistry, etc.), the CMS establishes an ordinal, 4-tier scale for assessing an individual's knowledge, skill, and experience. For each of the four tiers associated with a discipline, a set of criteria has been developed which an employee must meet in order to be assessed at that level.

Beaver and Schiavone conducted a study [BS06] in which development team skill was classified into categories based the CMS system described above. The percentage of the development team assigned to each skill category was then correlated to four different measures of software product quality. On the whole, the study found that an increasing presence of skilled software engineers had a positive impact on the quality of the delivered software product.

## 2.4 Software Product Quality Models

The term "software product quality model" refers to the assessment of an operational or fielded software product. That is, a software product quality model provides a framework for the assessment of software products at the time of delivery. These models are a very specific application of a software quality model and are simply for assessment. That is, they provide no predictive capabilities. However, the use of a comprehensive software product quality model is critical in software quality modeling because it provides a consistent approach to measuring the operational quality of a software product.

Surprisingly, given the vast amount of literature addressing the various aspects of software quality, there are few models that establish a framework for evaluating the quality of a software product. It is presumed that the ambiguous nature of many of the assessment criteria does not lend itself to be captured adequately in a quantitative model. However, the current software quality models have addressed both objective and subjective criteria as a basis for software quality determination.

J.A. McCall developed an early model for software product quality at the U.S. Air Force Rome Air Development Center in 1977 [MRW77]. McCall's quality model described a set of quality criteria that represented aspects of the software design. Derived from these criteria were software product qualities, which were caused by a set of the criteria. That is, a software product quality was derived from the combination of a subset of the quality criteria.

The ISO/IEC 9126 [ISO01] describes a model for evaluating software product quality. This model takes a very customer-centered approach to software quality and represents that quality in six categories: Functionality, Reliability, Usability, Efficiency, Maintainability, and Portability. The six ISO/IEC 9126 software quality attributes are defined in Table 2.4. Each of these categories reflects a unique area in customer satisfaction. The model's software quality attributes each include several sub-characteristics that create a hierarchy of quality factors and further define those particular aspects of software quality. The ISO/IEC 9126 establishes a framework for software product quality.

Table 2.4: ISO/IEC 9126 Software Quality Attributes [ISO01]

| Attribute | Definition |
|---|---|
| Functionality | The capability of the software product to provide functions that meets stated and implied needs when the software is used under specific conditions. |
| Reliability | The capability of the software product to maintain a specified level of performance when used under specified conditions. |
| Usability | The capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions. |
| Efficiency | The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions. |
| Maintainability | The capability of the software to be modified. |
| Portability | The capability of the software product to be transferred from one environment to another. |

The ISO/IEC 9126 model has been cited frequently in software quality research, and is recognized as a standard approach to assessing the quality of a software product. However, it is not without criticism. Kitchenham and Pfleeger [KP96] point out that there is a lack of rationale in the model as to what merits a software quality attribute, and what merits a sub-characteristic of that quality attribute. In addition, Kitchenham and Pfleeger argue that the relationship between quality attributes and sub-characteristics is not defined or rationalized.

That is, it is not clear that a given sub-characteristic is a more detailed derivation of an attribute. The correlation between the two qualities is not explained verbally in the model, or mathematically through relationships between indicator metrics.

Dromey, in a proposal of his own software product quality model [Dro95], argues that even though the quality attributes are decomposed in the ISO/IEC 9126, they are still too vague to provide adequate guidance in assessing the quality of a software product. Dromey takes a bottom-up approach to evaluating software quality in which structural elements of the software being developed have quality attributes that, when evaluated collectively, present a picture of total software product quality. The model provides a mapping between those quality aspects of each structural element and the higher-level quality attributes proposed in the ISO/IEC 9126. For example, Dromey argues that the correctness of the program statements are a common thread among all components of the software product, and affect both the functionality and reliability quality attributes identified in the ISO/IEC 9126 model. The author states in a follow-on paper [Dro96] that his model allows for a practical approach to building and assessing a quality product. In terms of assessing a software product under development, this may certainly be the case. However, the practicality of the Dromey model is limited to an assessment at the implementation phase of the software life cycle at the earliest. This approach does not lend itself well to the predictive aspect of software quality models.

The differences between the ISO/IEC 9126 and the Dromey Software Product Quality Models are largely in the mappings of product characteristics to quality attributes. The

ISO/IEC 9126 approach is a top-down hierarchy of attributes that culminates in a specific set of metrics that may be mapped to an overarching quality attribute. The Dromey model does not preclude the ISO/IEC 9126 model, but adds a different dimension by proposing threads of common quality attributes instead of specific metrics. For example, the quality attribute correctness has different interpretations depending on the software artifact being evaluated. However, the idea of completeness as a measure of how fully a given artifact addresses the software problem is a common thread across software development artifacts.

The need for a software product quality model as a complement to a predictive software quality model is obvious: the software product quality model provides the consistent framework for assessment and prediction. The nature of the software product quality model must accommodate both the objective and subjective measurements of software quality.

Software product quality models are a means to quantify the quality associated with a delivered software product. However, these models are simply mechanisms to assess a product at a very specific point in the development cycle. Software product quality models are not structured to explain how their measured quality attributes came about. That is, the causes of the quality of the software are not provided in a software product quality model, just a quantified representation of the quality of the software at the time of product delivery.

# CHAPTER 3

# METHODOLOGY

This chapter describes the research proposed for this doctoral dissertation. The intent is to advance the field of software engineering by improving the current approaches to modeling, evaluating, and predicting the quality of a software product under development. To this end, the scope of the research is described including stated goals and objectives that are intended to be met. The technical approach to modeling software is then detailed. The frameworks for measuring the causal factors and products of the software quality model are presented as well as the structure of the Bayesian Belief Network that relates those factors to products. This chapter concludes with a description of the approach to validation of the model, including the set of criteria for determining predictive validity.

## 3.1 Scope of Research

The following section establishes the scope of this research effort. The goals and objectives of the research are established. In addition, the applicability, relevance, and uniqueness of the research are addressed.

### *3.1.1 Research Goals and Objectives*

This research effort addresses a need for improved insight into the quality of software under development. The ability of a software project team to reliably assess and predict the quality of the software being developed provides significant technical and economic advantages. Technically, it provides a means for managing risk by identifying those areas in the development effort that are likely to be deficient in terms of the quality expectations of the customer or users. Economically, it allows the development team to take any necessary corrective actions earlier in the development life cycle, when corrections are least costly.

The goals of this research effort are to develop a model for assessing the quality of a delivered software product, to develop a model for predicting the quality of a software product under development, and to validate the model's accuracy of fit and predictive validity. When met, these three goals will satisfy the need for improved insight into the quality of software under development, and will result in the existence of a software quality product model that is useful to a software development team in terms of its ability to provide reliable assessments and predictions.

The objectives of this research effort are to develop a software quality model using Bayesian Belief Networks that captures the effects of problem complexity, process infrastructure, and development team skill/experience on software quality, to validate the developed software quality model's accuracy of fit by testing for equality of means and variances between predicted measures and actual software quality measures, and to validate the developed software quality model's predictive validity by testing the average relative error of the model's forecasts.

### *3.1.2 Relevance and Applicability*

Software development carries with it great risks in terms of both the quality of the delivered product, and meeting cost and schedule projections. The role of software quality is to maximize the correct operation, usability, and applicability to purpose of a developed software package, but in such a way that the development of such software remains feasible in terms of cost and schedule. To this end, software quality models are invaluable tools. They provide insight into the software development product and process that would otherwise go unnoticed. The visibility given through a quality model enables a project team to identify potential safety, cost, and schedule risks associated with the software under development, allows them to address those risks in a pre-emptive way, thus maximizing quality in the most cost effective way.

"Software quality" is a term that has a widespread meaning depending on one's perspective. A user may view software quality as a measure of the product's usability and flexibility, where a system engineer may view software quality as a measure of conformance to the specifications. Modeling software development accurately is a difficult endeavor due to the multifaceted nature of the software product and process. The complexity of the software coupled with measures that are either poorly defined, or lack any mathematical relationship to the product or process presents a unique challenge to the software engineering discipline. A model that could accurately represent the quality of a software product under development, based on inherent causal factors rather than environment-specific empirical relationships, would give a team remarkable predictive insight into their products, and allow for corrective action early. For example, the ability to identify early in development those aspects of the software that are most likely to be fault-prone would give a development team a tremendous opportunity to proactively respond to the model, and save cost and schedule.

### *3.1.3 Uniqueness of Research*

Software quality modeling is an area of software engineering research that has an established base of studies. Research that correlates design and implementation metrics to software quality measures (such as the number of faults) is commonplace in the field. Even the mechanism for quality modeling used in this research, Bayesian Belief Networks, was proposed ten years ago, in 1996. The following paragraphs detail the aspects of this research that separate it as a unique contribution to the field, and differentiate it from prior research efforts.

This research presents a software quality model that encompasses the entire software life cycle. Prior research, particularly research with Bayesian Belief Networks and software quality, has focused on particular aspects of the life cycle, or on specific relationships. No other known research has presented a model of the complete software life cycle. This research establishes causal ties between life cycle phases, and attempts to model the propagation of quality through the life cycle relative to an existing product quality standard.

This research effort also combines the quality causal factors of problem scope, personnel skill, and process infrastructure in its software quality model. Prior research has focused almost exclusively on the elements of the software problem, e.g. software size and complexity, as indicators of quality in a software product. This research recognizes the critical role that project personnel and process infrastructure play in producing a quality software product. There is no other known research effort that attempts to combine these categories of causal factors in a single software quality model.

## 3.2 Technical Approach

This section outlines the technical approach to the modeling of software quality. It begins with an overview (Section 3.2.1) of the methods used in representing and predicting quality in a software product. A justification/rationale is provided for the various modeling decisions, including the selected standards and industry in-use measurement frameworks (Section 3.2.3), and the decision to use Bayesian Belief Networks as the modeling mechanism (Section 3.2.4). Section 3.2.5 describes the possible options for model structure that are explored, and the details of the cause-effect relationships between model variables.

### *3.2.1 Technical Overview*

This research proposes a model to predict software product quality. Developing a software quality model is challenging for several reasons. The quantity of information that has to be taken into consideration in a software development effort is overwhelming as there are many factors that could possibly influence the final quality of a software product. The model developed for this research accounts for influences in the categories of development team skill, process maturity, and problem complexity. Figure 3.1 depicts at a high level the elements used as inputs and outputs of the model. The model takes as inputs measurements of development team skill, software process maturity, and software problem complexity over the entire software life cycle, and produces as an output forecasts of software product quality.

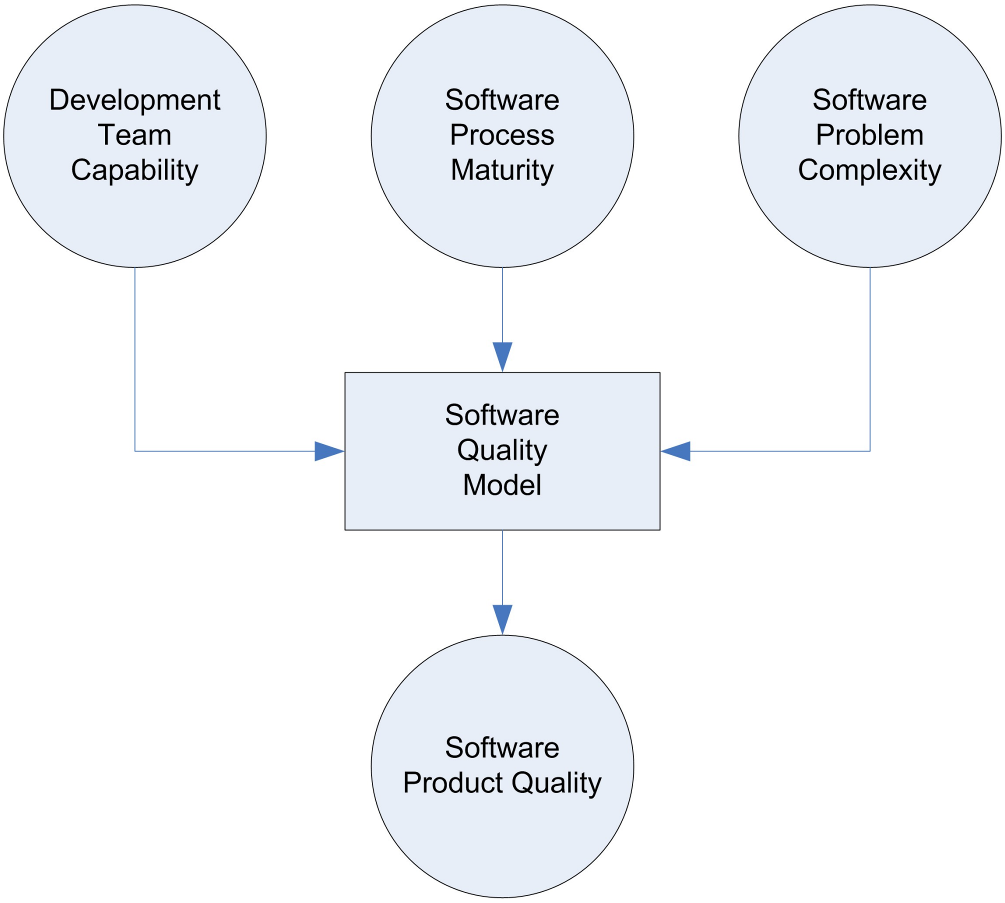
Development Team Capability
Software Process Maturity
Software Problem Complexity
Software Quality Model
Software Product Quality

Figure 3.1: High-Level Diagram for Modeling Software Product Quality

The approach to developing a software quality model for this research involved answering several questions:

1. What kind of measures would be used for model inputs and outputs?

2. What modeling technique would be used to relate inputs to outputs?

3. How would the selected measures be represented and related within the model?

The answers to these questions define the methodology used in this research.

In terms of the measures used to define the inputs and outputs to the model, the intent was to use existing measurement standards or industry in-use frameworks to characterize the model inputs and outputs. The primary advantage to this approach is that the model leverages existing bodies of work in characterizing elements of software skill, process, and complexity instead of formulating measures with no academic or industry pedigree. Section 3.2.3 details the specific measurement standards and frameworks that were selected, and provides a rationale for their selection. In addition to making the developed model more useful with respect to its ability to incorporate credible software engineering measures, employing standards as the basis for measurement also makes the developed model more marketable to software development organizations. Instead of needing to adopt a customized measurement program to use the quality model produced by this research, a development organization will likely already have institutionalized many of the necessary measures.

The approach to selecting the modeling technique was to adopt a method that was validated in industry, can adapt to the environment of a specific development organization, is understandable in its structure, and can operate in the presence of missing data. The Bayesian Belief Network (BBN) was selected to model software quality in this research, and the rationale for that decision is detailed in Section 3.2.4. BBNs excel at representing cause-effect relationships between variables, and are capable of performing in the presence of missing inputs. The flexibility of the BBN is also a key component to the usefulness of the model. If a development organization needs to institute a complete measurement program in order to use the developed software quality model, then that model becomes much less attractive to use because it is such a significant investment. However, if the model can adjust to the environment of the development organization, including a measurement program that may only be a subset of what the model can handle, then the model becomes much more feasible to implement.

The implementation of the model (captured in Section 3.2.5) describes the way in which the selected measurement frameworks and the selected modeling technique are combined to create an accurate software quality model. This includes a specification of the manner in which each measurement is represented in the proposed model structure. This research identifies three possible BBN structures that will relate the inputs to the outputs, with the structures being compared later in Section 4.4. Also, the tools used to implement the software quality model are identified.

### *3.2.2 Technical Assumptions*

The software quality model proposed in this research assumes that the development team is performing the following four phases of the software engineering life cycle, executed in sequential order: Requirements, Design, Implementation, and Integration/Test. These phases represent the minimum engineering approach to software development.

Another assumption in the development of this model is that software quality propagates through the development life cycle. This assumption is logically inferred, but not proved, and attempts to identify a ”snowball effect”. That is, this research assumes that the level of quality of upstream activities affects the level of quality in downstream activities. For example, poorly captured requirements will logically affect the quality of the design solution. This assumption is captured in the model by those causal relationships established between life cycle phases (see Section 3.2.5.2).

### *3.2.3 Measurement Framework Selection*

Modeling software quality requires the use of several measurement frameworks to consistently quantify the proposed model inputs and outputs. There are four high-level elements of the proposed software quality model that require a consistent approach to measurement, and are described in this section: development team skill (Section 3.2.3.1), software process maturity (Section 3.2.3.2), software problem complexity (Section 3.2.3.3), and software product quality (Section 3.2.3.4). Each of these four selected measurement frameworks is based on existing standards and/or widely accepted approaches to measurement.

#### *3.2.3.1 Measuring Development Team Skill*

The assessment of the capability of the software development team requires insight into the team's collective education and experience. The assumption with this approach is that a more educated and experienced team will produce higher quality in their software product and artifacts. The software life cycle is comprised of four phases, each of which requires different types of skills/experience to accomplish successfully. The requirements phase is focused on organizational and interpersonal skills: interacting with customers, eliciting needs from those customers, and organizing the needs into a set of requirements. The design phase requires the ability to create a software solution that meets the identified requirements, and skills in evaluating the set of approaches and tools to implement that solution. Implementation demands a working knowledge and skill in the specific development tools, including languages, operating systems, libraries, and development environments. Integration and Test involves foresight in verifying the spectrum of possible operational and failure scenarios, and the knowledge/skills to determine an appropriate level of testing for a given product and a given use.

Because of the wide range of skills required at each life cycle phase, the approach to appraisal is to evaluate skill and experience of the team in terms of each of the life cycle phases: Requirements, Design, Implementation, and Integration/Test. It is important to capture skills at each life cycle stage so that a correct picture of the team's capabilities is portrayed. For example, a development team with vast design skill but little requirements development skill may produce a fantastic product, but may not necessarily meet the needs of the customer. Each team member will be rated individually based on their skill and experience in each of the life cycle phases. The individual skill ratings will be consolidated into a set of measures that reflects the distribution of skills for the entire team across the life cycle phases.

This research has adopted the CMS [NAS05] (See Section 2.3.3) approach to individual skill assessment, and applied it in terms of the software engineering life cycle. A four-tier ordinal scale is used to rate an individual's skill/experience within each of the four major phases of the life cycle: requirements, design, implementation, and integration/test. Associated with each tier, and for each phase, is a set of criteria which, if met, would allow an individual to be assessed the appropriate numerical rating for that phase.

Although NASA's CMS provides the basis for an approach to individual skill assessment, it is necessary to broaden the individual ratings in order to capture the combined skill and experience of an entire software development team. Assessing skill for a development team involves consolidating the individual skill levels into a set of measurements that would accurately reflect the team's skill distribution in each of the life cycle phases. Table 3.1 below lists the 16 metrics selected to represent a development team's skill and experience. For each phase and skill level, the percentage of the development team members that were assessed at that phase/skill level is the metric used. Normalizing the distribution of team skill by team size allows for the comparison of projects regardless of the number of team members. In addition, incorporating measures that independently address each life cycle phase allows for different team sizes and skill mixes in each phase to be accurately represented in the model. Within the software quality model, the percentage of team members with a given skill level at a given life cycle phase was represented discretely in four categories. Each category represents a quarter of the percentage spectrum.

Table 3.1: Software Development Team Skill/Experience Factors.

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

The approach used in this research for the assessment of development team skill and experience is to quantify individual skill and consolidate that into a measure of development team skill for each software life cycle phase. No attempt is made in this research to analyze the composition of the development team in terms of personality, or in the light of team dynamics. This perspective on collective team skill is very complex and difficult to assess, and is beyond the scope of this research effort. However, investigating the effect of development team cohesiveness on software product quality seems a logical extension of this work, and a solid candidate for future research.

#### *3.2.3.2 Measuring Software Process Maturity*

The assessment of the process maturity is the quantification of an organization's ability to internally identify, elevate, and instill those best practices that lead to repeated successful results. The organization's approach to the activities associated with eliciting requirements, creating a design, implementing the design and verifying/validating the implementation have a direct effect on the quality of the software being produced. At a practical level, this approach includes of the assets in the organization's infrastructure that enable software projects to succeed: processes, tools, and templates.

The ISO/IEC 15504: Software Process Assessment [ISO98] is an international standard for assessing software process maturity, and was the basis for quantifying the development processes in this research. While several other software process models were considered (See Section 2.3.1), they are primarily methods of software process improvement, with process assessment as a secondary objective. That is, the other models focus on providing a path for systematically improving an organization's development infrastructure with process assessment merely a means for measuring the progress of the improvements. The ISO/IEC 15504 is a standard focused specifically on process assessment, without bias to a particular model for improvement, and is the product of a collaboration of all of the process improvement models. Thus, it is a logical choice for the purposes of process assessment in this research, which should not be biased by any specific improvement model.

The ISO/IEC 15504 is comprised of several different process categories, processes, and practices which serve to address all activities associated with a software engineering project. Only a subset of the ISO/IEC 15504 processes, contained in the Engineering process category, are applicable to the four traditional phases of the software development life cycle: requirements, design, implementation, and integration/test. Table 3.2 lists the five ISO/IEC 15504 processes, and their associated 27 practices, that were quantified for this research. Collectively, the 27 practices are the set of activities that characterize a mature software development organization.

Each practice is represented in the software engineering data as a dichotomous variable that indicates a software project's compliance or non-compliance with the practice. For example, a software project that meets with the customers during the requirements phase to evaluate the correctness and completeness of the software specification will be assigned a rating of ıtrue for the ”Evaluate/Validate Requirement With Customer” practice. The binary approach to assessment is a break from the 5-tier scale used by the ISO/IEC 15504 standard to indicate a project's level of compliance. This more simplified approach was selected due to a lack of resources to perform a formal assessment on each software project, and to reduce the state space associated with the software quality model itself. Process Maturity is the only measure in the software engineering data that is quantified discretely.

Table 3.2: Software Process Maturity Factors

| ISO/IEC 15504 Process | Practices |
|---|---|
| Software Requirements Analysis | Specify Software Requirements<br>Determine Operating Environment Impact<br>Evaluate/Validate Requirement With Customer<br>Develop Validation Criteria for Software<br>Develop Release Strategy<br>Update Requirements<br>Communicate Software Requirements<br>Evaluate Software Requirements |
| Software Design | Develop Software Architectural Design<br>Design Interfaces<br>Verify the Software Design<br>Develop Detailed Design<br>Establish Traceability |
| Software Construction | Develop Software Units<br>Develop Unit Verification Procedures<br>Verify the Software Units<br>Establish Traceability |
| Software Integration | Develop Software Integration Strategy<br>Develop Integrated Software Item Integration Strategy<br>Develop Tests for Integrated Software Items<br>Test Integrated Software Items<br>Integrate Software Item<br>Regression Test Integrated Software Items |
| Software Testing | Develop Integrated Software Test Strategy<br>Develop Tests for Integrated Software<br>Test Integrated Software<br>Regression Test Integrated Software |

However, the use of the correlation coefficient as a means of establishing a linear relationship between variables (see Section 3.3.2.2) is not threatened by this combination of measurement scales. Correlating a dichotomous (or binary) discrete measure to an interval or ratio scale variable is a special case correlation called a Point-Biserial Coefficient of Correlation, and for this type of correlation, it is acceptable to use the dichotomous variable like a continuous variable (that is, represented as a continuous 1 or 0) in the correlation calculation [Sci06] [Gar06].

Table 3.3: Life Cycle Effects of Software Problem Complexity

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

#### *3.2.3.3 Measuring Problem Complexity*

Software problem complexity is a concept that has different implications throughout the development life cycle. The challenge in assessing problem complexity is to identify the measures that capture the way in which complexity manifests itself in each given phase. Initially, the complexity of a software project embodied in the need statements, including quality needs, which have been elicited from the customer or user community. From customer needs, the complexity is expanded to include the intricacy of the design solution, method of implementation, coverage of the testing, and volatility of the various project artifacts. Table 3.3 summarizes those questions which, when answered, provide insight into the effect of software problem complexity on each life cycle phase.

In the requirements phase, the development team attempts to understand and formalize both the operational and quality needs of the customer. Complexity in the requirements phase is an extension of the customer needs, and captures whether the developed specification addresses those identified needs. For this research, the initial level of complexity associated with a software development effort is represented as a cross-section of customer needs based on the quality characteristics detailed in the ISO/IEC 9126 software product quality standard [ISO01] (See Section 2.4). Table 3.4 below details the spectrum of quality needs that were considered. A total of 34 metrics were analyzed and modeled in an attempt to capture complexity in terms of the customer's needs. In the proposed software quality model, a Boolean metric exists for each software quality need which indicates whether or not that particular need was adequately covered in the software specification.

Complexity in the requirements phase may also be captured by the number of changes associated with the specification. A high volume of changes may indicate poor understanding of the needs of the customer, and can affect the quality of the software product. The model attempts to account for this factor by tracking the percentage of requirements that are changed after the specification has been approved, and the software project has progressed to software design.

The design phase is the point in the software life cycle where the development team forms the technical approach to meeting the customer's requirements. Complexity in the design phase of the software life cycle is concerned more with the system architecture, the amount interaction between its components, and the degree to which the design has been partitioned for implementation. Table 3.5 below summarizes the design complexity measures used in this research. These measures of design complexity are intended to capture the extent to which the design solution for the system evolved within the design phase. As in the requirements

Table 3.4: Software Quality Needs

| ISO/IEC 9126 | |
|---|---|
| **Quality Characteristic:** | |
| Quality Indicator Metrics | Needs Description |
| **Functionality:** | |
| Functional Adequacy | Were there customer needs in terms of functional operation? |
| Functional Completeness | Were the customer needs captured completely? |
| Functional Correctness | Were the customer needs implemented correctly? |
| Specification Stability | Were the customer needs stable as development progressed? |
| Computational Accuracy | Were there customer needs for accuracy in the results? |
| Precision | Were there needs for precision in the computed results? |
| Data Exchangeability | Were there customer needs for custom interface formats? |
| Interface Consistency | Were there customer needs for custom interface protocols? |
| Access Auditability | Were there customer needs for auditing user access? |
| Access Controllability | Were there customer needs for controlling user access? |
| Data Corruption Prevent | Were there customer needs for preventing data corruption? |
| Data Encryption | Were there customer needs in terms of encrypting data items? |
| Functional Compliance | Were there customer needs in terms of functional standards? |
| **Efficiency:** | |
| Time Behavior | Were there customer needs for specific time behaviors? |
| I/O Utilization | Were there needs for budgeting input/output resources? |
| Memory Utilization | Were there customer needs for budgeting memory resources? |
| Efficiency Compliance | Were there customer needs in terms of performance standards? |
| **Reliability:** | |
| Failure Avoidance | Were there customer needs to actively avoid failures? |
| Incorrect Op Avoidance | Were there customer needs to handle incorrect operation? |
| Restorability | Were there customer needs to restore the system after failure? |
| Restore Effectiveness | Were there customer needs for a system restoration time? |
| Reliability Compliance | Were there customer needs in terms of reliability standards? |
| **Usability:** | |
| User Op Cancelability | Were there customer needs to cancel user operations? |
| User Op Undoability | Were there customer needs to undo user operations? |
| Interface Customizability | Were there expectations to customize the interface? |
| Physical Accessibility | Were there customer needs for accommodating disabilities? |
| Op Status Monitoring | Were there customer needs for monitoring the system's status? |
| Op Error Recoverability | Were there needs for recovery from operational errors? |
| Usability Compliance | Were there customer needs in terms of usability standards? |
| **Maintainability:** | |
| Activity Recording | Were there customer needs for recording system activities? |
| Diagnostic Functions | Were there customer needs for software diagnostic functions? |
| Maintainability Compliance | Were there customer needs in terms of maintainability standards? |
| **Portability:** | |
| Hardware Adaptability | Were there needs for operation across hardware environments? |
| Software Adaptability | Were there needs for operation across software environments? |

Table 3.5: Software Design Complexity Metrics

| Metric | Description | Calculation |
|---|---|---|
| Depth of Inheritance Tree | Measures the depth of the class hierarchy. | $X = A$, where A is the longest root-leaf distance in the class tree. |
| Design Expansion | The degree to which the architectural design is expanded to a detailed design. | $X = \frac{A}{B}$, where<br>A = # design components<br>B = # designed classes |
| Interface Protocol Expansion | Indicator of the complexity of the implemented interface protocols. | $X = \frac{A}{B}$, where<br>A = # interfaces<br>B = # interface protocols |
| Interface Format Expansion | Indicator of the complexity of the implemented interface protocols. | $X = \frac{A}{B}$, where<br>A = # interfaces<br>B = # interface formats |

phase, another characterization of design complexity is the number of changes that are made to the software design. After the design has been accepted, a large number of changes may have an affect on the ultimate quality of the software product.

Problem complexity is represented in software implementation as the degree to which captured requirements affect the source code. That is, what proportion of the software implements an expressed quality need (e.g., Accuracy, Precision, etc). This measure, called prevalence, is modeled as the ratio of the number of source code units that implement a given quality need to the number of implemented source code units. The intent of this measure for each quality need is to indicate the complexity that the need has caused in the implemented product.

Software integration and test is the life cycle phase in which the developed source code is verified to operate correctly as a system. Problem complexity is represented in this phase as the extent to which the software requirements and customer needs are covered in the verification and validation of the system. Numerically, this is expressed as the ratio of the number of requirements verified and validated in the integration and test phase to the number of requirements.

#### *3.2.3.4 Measuring Software Product Quality*

The assessment framework chosen to measure the quality of a software product at the time of delivery is the ISO/IEC 9126 standard. This standard, discussed in depth in Section 2.4, represents the most widely used and mature standard for assessing the quality of a software product. The use of a software product quality model is essential to this research because all software quality assessment and prediction activities will be expressed in terms of the product quality at the time of delivery. Due to the dependent nature of the software life cycle, quality (or lack of quality) at any phase propagates to the next phase. The software product quality serves as a consistent point of measurement in the model. This assessment of product quality is based on the available data, and is not necessarily comprehensive depending on the current phase of the life cycle. For example, if a software development effort is in the Requirements Phase, the projected software quality of the product may be high, but at the time of assessment it is still unknown. Ideally, the model will converge on the actual measures of software product quality as the software development effort progresses.

### *3.2.4 Modeling Technique Selection*

This research proposes the use of Bayesian Belief Networks as the mechanism for representing causal relationships in software development, and relating specific measures of software product, process and development team quality to software product quality data. This approach requires a mature metrics program in order to capture the data necessary to accurately represent the underlying software development process. Using a versatile set of assessment frameworks that capture team capability, process capability, and software problem difficulty, a history of data is compiled that allows the application of Bayesian Belief Networks to validate causal factors in software development. The result is a model of the software development process that identifies those elements of the software under development that are likely to be at risk for low quality in terms of the ISO/IEC 9126 software product quality standard.

#### *3.2.4.1 Detailed Description of Bayesian Belief Networks*

The Bayesian Belief Network (BBN) is the mathematical technique selected to model software quality in this research effort. A BBN is an adaptive system, meaning that its underlying algorithm allows for the adjustment of model output values based on prior data sets. A Bayesian Belief Network is graphically represented as a collection of nodes that are connected by directed arcs. The nodes that comprise a BBN represent random variables within the model, with each node providing probabilities of outcomes based on a set of input values. The directed arcs in a BBN represent causal relationships or dependencies between nodes. The mathematical function that governs each BBN node is Bayes' Theorem.

Bayes' Theorem is a set of conditional probability equations based on the propositions developed by Reverend Thomas Bayes in 1763, and published posthumously in the "Essay Towards Solving a Problem in the Doctrine of Chances" [Bay63]. In his essay, Bayes postulated the relationship between the probabilities of two sequential events as the ratio of the compounded probabilities of each. The resultant Bayes equation is depicted in the Equation 3.1.

**Bayes' Rule** [MS95]

Given k mutually exclusive and exhaustive states of nature (events), $A_1, A_2, \ldots, A_k$, and an observed event E, then $P(A_i|E)$, for $i = 1, 2, \ldots, k$, is

$$P(A_i|E) = \frac{P(A_i \cap E)}{P(E)} = \frac{P(A_i, E)}{P(E)} = \frac{P(E|A_i) * P(A_i)}{P(E)} \tag{3.1}$$

The left hand side of this equation, called the "posterior probability", represents the probability the event $A_i$ has occurred given the evidence $E$. The term $P(E|A_i)$ in the equation is called the "likelihood" and represents the probability that the evidence E exists in the presence of event $A_i$. The "prior probability", represented in the term $P(A_i)$, is the probability the event $A_i$ would happen before the evidence $E$ was introduced. The final term, $P(E)$ is the independent probability that the evidence would occur. Although Bayes' work was an important contribution to mathematics, it was not applied as part of a model of causality until the mid-1980's. At the time of Reverend Bayes, the idea of causality and mathematically representing cause and effect relationships was still in its infancy.

Sewall Wright, in 1921 [Wri21], proposed the first network used to represent causation in a system, and to propagate probabilistic information. Wright's application and validation consisted of an analysis of both the birth weight of guinea pigs, and the transpiration of plants. Wright's networks were depicted as diagrams in which variables were arranged in the figure, and arrows were used to indicate a cause-effect relationship. The networks proposed by Wright used systems of equations to mathematically represent the causal relationships and propagate probabilities through the network. Causal networks are a way of showing a chain of cause-effect relationships throughout a series of nodes. Each node in a causal network represents an event in time. Thus, the original intent of the causal network was to represent a sequence of events.

The format for the causal network is a directed acyclic graph (DAG). Consider the following fictitious scenario that will be analyzed and modeled as an example of how a causal network emulates logical decisions.

> *A supervisor has two employees, John and Susan. Upon learning that John is late, the supervisor presumes that bad traffic has held up the wayward employee. He passes by Susan's desk to discover her there working diligently.*
>
> *The supervisor asks, Ẅere you at work on time today, Susan?¨*
>
> *”Yes sir, why do you ask?” she replies.*

*"We should cancel the 9am meeting", retorts the supervisor,*

*"John has overslept again".*

This scenario can be easily represented in a causal network. Consider the representation of this scenario formalized in Figure 3.2 below. There are four variables introduced: John is late (J), Susan is late (S), Traffic is bad (T), and John has overslept (O). Each of these variables has two possible states: true or false. In this causal network, bad traffic (T) affects the certainty that both Susan (S) and John (J) will be late. In this case, John is late (J = *true*), and Susan is on time (S = *false*). Because Susan is present, the supervisor's certainty that John was late due to bad traffic was decreased (T = *false*), and his certainty that John was late due to oversleeping was increased. The supervisor inferred that John had simply overslept (O = *true*).

This example demonstrates the relationship between dependent and independent variables in causal networks. Initially the causal network implies a dependence of John's tardiness (J) and Susan's tardiness (S) on the state of the traffic (T), and John's alarm clock (O). However, the information obtained from the scenario changed the dependence of the variables. The knowledge that John was late and Susan was not late affected the certainty of the causal factors; in effect, making them dependent on the information that was available at the time. Information that is known for certain and is not affected by the state of other information is called independent. Information whose certainty can change based on other available information is called conditionally independent.

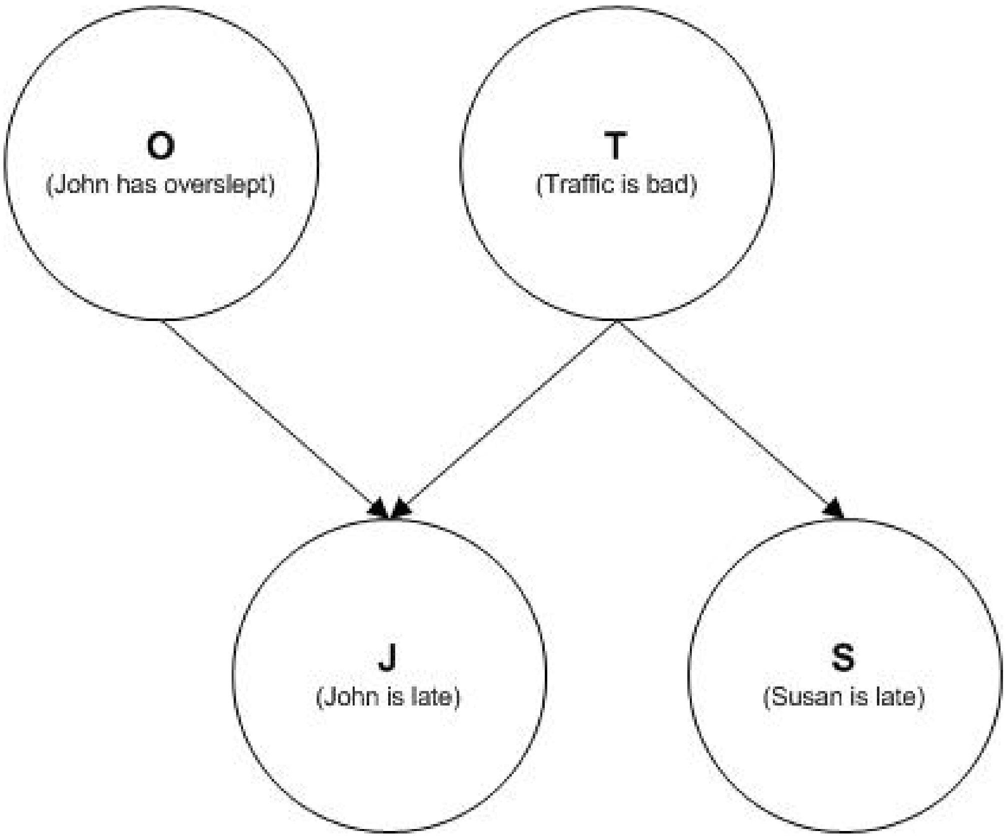

Figure 3.2: A Causal Network Representing the Scenario

Casual networks, in practice, are similar to the example provided in Figure 3.2. They have nodes that represent events, and directed links that represent a cause-effect relationship between events. The example used nodes with two states (*true*, *false*) where, in practice, a node can have as many states as necessary, or even be continuous. A causal network with a quantifiable number representing certainty of an event is a special case called a Bayesian Belief Network.

In 1986, Judea Pearl [Pea86] expanded on the idea of causal networks by introducing belief networks. Belief networks are causal networks in which the relationships between nodes (represented as directed arcs) are assigned probabilities. The probability of the outcomes of a given node is calculated based on the probabilities assigned to the input arcs to that node. This value of probability is a means for representing the certainty associated with a particular outcome. The method of calculation for model inference is Bayes' Rule of conditional probability.

In a Bayesian Belief Network, the initial calculations of probabilities as they propagate through the network are determined through the joint probability distributions of each variable. This joint probability distribution is the product of all relevant conditional probabilities for a given node. It is characterized by Equation 3.2.

**Joint Probability Distribution (JPD) [Jen96]**

Let BN be a Bayesian Network over $U = A_1, \ldots, A_m$. Then the Joint Probability Distribution P(U) is the product of all conditional probabilities in BN:

$$P(U) = \prod{}_i P(A_i | pa(A_i)) \tag{3.2}$$

Where pa($A_i$) is the parent set of $A_i$.

Consider the scenario above as a Bayesian Belief Network. Assume that the probability for heavy traffic is 40% (P(T) = 0.4), and the probability that John has overslept is 20% (P(O) = 0.2). Also, assume that both employees are late 50% of the time when traffic is bad, and that John is always late when he oversleeps. Although graphically, the network remains the same, the Bayesian version now has probability values associated with it. Table 3.6 details how the joint probability distribution definition is applied to determine initial probability that Susan is late.

Table 3.6: Joint Probability Distribution to Determine P(S = *true*) and P(S = *false*)

|  | Susan is Late |  | JPD | JPD |
|---|---|---|---|---|
| Traffic is Bad (T) | Yes | No | (Susan is late) | (Susan is not late) |
| P(T = *true*) = 0.4 | 0.5 | 0.5 | P(S = *true*) = 0.2 | P(S = *false*) = 0.2 |
| P(T = *false*) = 0.6 | 0.0 | 1.0 | P(S = *true*) = 0.0 | P(S = *false*) = 0.6 |
|  |  | Total: | P(S = *true*) = 0.2 | P(S = *false*) = 0.8 |

Using the definition of Joint Probability Distribution in Equation 3.2, the Joint Distribution is determined by multiplying the conditional probabilities. In this case, the initial probability that Susan is late is calculated by multiplying the probability that traffic is bad by the associated conditional probability that Susan will be late (the "Yes" column). The independent results are then summed to provide the total probability. The calculations reveal that before any information is known, the probability that Susan will be late to work is 0.2.

Table 3.7 details a similar calculation to determine the initial probability that John will be late. This Table takes into consideration two causal factors to John's tardiness: oversleeping and bad traffic.

Table 3.7: Joint Probability Distribution to Determine P(J = *true*) and P(J = *false*)

|  |  | John is Late |  | JPD | JPD |
|---|---|---|---|---|---|
| Traffic is Bad (T) | John Overslept (O) | Yes | No | (John is late) | (John is not late) |
| P(T = *true*) = 0.4 | P(O = *true*) = 0.2 | 1.0 | 0.0 | P(J = *true*) = 0.08 | P(J = *false*) = 0.0 |
| P(T = *true*) = 0.4 | P(O = *false*) = 0.8 | 0.5 | 0.5 | P(J = *true*) = 0.16 | P(J = *false*) = 0.16 |
| P(T = *false*) = 0.6 | P(O = *true*) = 0.8 | 1.0 | 0.0 | P(J = *true*) = 0.12 | P(J = *false*) = 0.0 |
| P(T = *false*) = 0.6 | P(O = *false*) = 0.8 | 0.0 | 1.0 | P(J = *true*) = 0.0 | P(J = *false*) = 0.48 |
|  |  |  | Total: | P(J = *true*) = 0.36 | P(J = *false*) = 0.64 |

Again, the joint distribution is determined by multiplying the conditional probabilities. In the case of John, two causal factors are considered, and reflected in the probabilities that

John is late. Since John is always late whenever he oversleeps, the probability for John to be late is 1.0 when O = *true*. However, when John has not overslept, his tardiness depends on the state of traffic, as depicted in the probability for him to be late under those conditions. Given the system with the known conditional probabilities, the probability that John will be late on any given day is 0.36.

To continue the example, consider the rationale of the supervisor in assessing that John has overslept. This is modeled in the Bayesian Belief network as a set of information about the tardiness of each employee, and a calculated probability of the occurrence of each causal factor. Bayesian Belief Networks use Bayes' Rule to handle inference in the network. That is, a level of confidence can be assigned to each unknown data point in the network based on the observed states of each known data point. In this case, the supervisor has two pieces of information: John is late (J = *true*), and Susan is not late (S = *false*). Consider each event as it occurred.

The supervisor initially learns that John is late. This is an item of evidence in the Bayesian Belief Network that alters the probabilities associated with each of the other nodes in the network. Bayes' Rule is used to determine the updated probabilities based on the new evidence. Equation 3.3 and Equation 3.4 shows the calculations of P(T = *true* | J = *true*), the probability that traffic is bad given that John is late, and P(O = *true* | J = *true*), the probability that John overslept given that John is late.

$$P(T = true|J = true) = \frac{P(J = true, T = true)}{P(J = true)} = \frac{0.08 + 0.16}{0.32} = 0.66667 \qquad (3.3)$$

$$P(O = true|J = true) = \frac{P(J = true, T = true)}{P(O = true)} = \frac{0.08 + 0.12}{0.32} = 0.55556 \qquad (3.4)$$

The calculation to determine the updated probability of bad traffic is made using the joint probability of the events that John is late and that traffic is bad and dividing by the overall probability that John is late. The joint probability table to determine John's probability for being late (See Table 3.7) shows two instances in which both John is late (J = *true*) and Traffic is Bad (T = *true*). The sum of these two conditional probabilities represents the joint probability of these events. The resultant probability that Traffic is bad given that John is late is 0.66667. A similar calculation is performed to determine the probability that John overslept is 0.55556. From the experience of the supervisor, knowing only that John is late, it is presumed that John was caught in bad traffic. The Bayesian Belief Network correctly models this by reflecting a higher probability that traffic is bad, P(T = *true*) = 0.66667, than that John overslept, P(O = *true*) = 0.55556.

Before proceeding to the next evidential development in the scenario, it is important to demonstrate how the introduction of evidence changes the values in the joint probability tables. The application of Bayes' Rule to each of the elements of the joint probability distributions for John's lateness (variable J) is necessary to reflect the updated probabilities.

Equation 3.5 below provides an example of how Bayes' Rule is applied to a specific joint probability, and Table 3.8 below is the Joint Probability Table updated for all elements.

$$P(T = true, O = true|J = true) = \frac{P(J = true, T = true, O = true)}{P(J = true)} = \frac{0.08}{0.36} = 0.2222 \quad (3.5)$$

Notice that the updated Joint Probability Table reflects the evidence that is now known: that John is late. Thus, the probability that John is on time (J = *false*) is 0.0 for all conditions. Given that this event has occurred, and future calculations employing the Joint Probability Distribution associated with variable J will use the updated conditional probabilities stored in Table 3.8.

Table 3.8: Joint Probability Distribution for Variable J (J = *true*)

| Traffic is Bad (T) | John Overslept (O) | JPD<br>(John is late) | JPD<br>(John is not late) |
|---|---|---|---|
| P(T = *true*) | P(O = *true*) | P(J = *true*) = 0.2222 | P(J = *false*) = 0.0 |
| P(T = *true*) | P(O = *false*) | P(J = *true*) = 0.4444 | P(J = *false*) = 0.0 |
| P(T = *false*) | P(O = *true*) | P(J = *true*) = 0.3333 | P(J = *false*) = 0.0 |
| P(T = *false*) | P(O = *false*) | P(J = *true*) = 0.0 | P(J = *false*) = 0.0 |
| | Total: | P(J = *true*) = 1.00 | P(J = *false*) = 0.00 |

In addition to resolving how the conditional probabilities associated with John's tardiness are affected by the evidence that he is late, the effect on Susan' likelihood to be tardy must also be considered. Knowing that John is late increases the belief that traffic is bad on this particular day. This also increases the likelihood that Susan will be late. The original Joint Probability Distribution (shown in Table 3.6) for variable S uses the initial values for variable T. This Table must be updated to reflect the new values for variable T given that John is now known to be late. Table 3.9 below captures this update. Knowing that John is late increases the likelihood that Susan is late. As with variable J, future calculations employing the Joint Probability Distribution associated with variable S will use the updated conditional probabilities stored in Table 3.9.

Table 3.9: Updated Joint Probability Distribution for Variable S (J = *true*)

| | Susan is Late | | JPD | JPD |
|---|---|---|---|---|
| Traffic is Bad (T) | Yes | No | (Susan is late) | (Susan is not late) |
| P(T = *true*) = 0.66667 | 0.5 | 0.5 | P(S = *true*) = 0.33333 | P(S = *false*) = 0.33333 |
| P(T = *false*) = 0.33333 | 0.0 | 1.0 | P(S = *true*) = 0.0 | P(S = *false*) = 0.33333 |
| | | Total: | P(S = *true*) = 0.33333 | P(S = *false*) = 0.66667 |

Consider now the changes in the supervisor's belief upon discovering that Susan was on time to work. This event injects another item of evidence into the network, and therefore affects the probabilities associated with each unknown data point that is connected. Since Susan's punctuality is affected only by the traffic, Bayes' Rule is applied in Equation 3.6 to calculate P(T = *true* | S = *false*), the new probability that traffic is bad given that Susan is on time.

$$P(T|S = \mathit{false}) = \frac{P(S = \mathit{false}, T = \mathit{true})}{P(S = \mathit{false})} = \frac{0.3333}{0.6667} = 0.5 \tag{3.6}$$

The conditional probability P(S = *false*, T = *true*) elements in this calculation are taken from the updated Joint Probability Distribution table for variable S (See Table 3.9). Similar to the procedure for variable J above, the joint probabilities in the variable J are adjusted to reflect the change in the probability of the variable T. The resultant Joint Probability Table is shown in Table 3.10.

Table 3.10: Updated Joint Probability (S = *false*)

| Traffic is Bad (T) | John Overslept (O) | JPD<br>(John is late) | JPD<br>(John is not late) |
|---|---|---|---|
| P(T = *true*) | P(O = *true*) | P(J = *true*) = 0.1666 | P(J = *false*) = 0.0 |
| P(T = *true*) | P(O = *false*) | P(J = *true*) = 0.3333 | P(J = *false*) = 0.0 |
| P(T = *false*) | P(O = *true*) | P(J = *true*) = 0.5 | P(J = *false*) = 0.0 |
| P(T = *false*) | P(O = *false*) | P(J = *true*) = 0.0 | P(J = *false*) = 0.0 |
| | Total: | P(J = *true*) = 1.00 | P(J = *false*) = 0.00 |

Based on the updates, the probability of John oversleeping can be calculated to P(O = *true*) = 0.66667. Thus, this Bayesian Belief Network has successfully modeled the reasoning of the supervisor in deducing that John had overslept, P(O = *true*) = 0.66667, over the likelihood of bad traffic, P(T = *true*) = 0.5. When observed evidence is incorporated into the network structure, the resultant probabilities for causes of the observed evidence are quantifiably clear.

It is this modeling of reason and belief that is intended to be applied to software quality as events unfold in the software development process. The model is intended to follow the time sequence associated with the classic waterfall life cycle model, and reflect probabilities associated with different quality outcomes based on the measurements (or evidence) introduced into the model.

#### *3.2.4.2 Justification of Selection of Bayesian Belief Networks*

Bayesian Belief Networks have several advantages over other complex adaptive systems with respect to modeling the quality of a software development effort. BBNs are capable of representing cause-and-effect relationships using both subjective and objetive data, can adapt to prior data, and can infer unknown data elements based on a set of known data elements. Although neural networks also posses these qualities, neural networks are not as expressive. Each node in a BBN represents an independent proposition within the network. Intermediary nodes in a neural network have no such concrete meaning. In addition, BBNs can handle providing a range of values for a given prepositional output, and a probability associated with each value. Neural networks conversely can provide only a designated output value (discrete or continuous), and cannot handle multiple possible outcomes to a proposition.

To date, there have been numerous successful applications of Bayesian Belief Networks across a variety of industries. BBNs are used in medicine to assist in the monitoring of patients in intensive care [BSC89], as an aid in the diagnosis of lymph node diseases [HHH92], and for ovarian tumor classification [AFT03]. They have also been applied in robot sonar mapping [BS97], and in uncollected debt detection [ES95]. Most notably, they are used as the engine behind the office assistant in Microsoft Office products [HH98].

The structure of a BBN can cause difficulties in terms of performance. The size of the conditional probability tables grow exponentially with the number of arcs between nodes, particularly when multiple arcs are directed to the same node. For this reason, complex

BBNs are difficult to construct without maximizing the resources of a computer. Although this issue is a concern, it was decided to go forward with BBNs as the modeling technique, and provide a commentary on their suitability to the problem of modeling software quality in Chapter 5.

### *3.2.5 Model Implementation*

Model implementation is the process of providing a computational context for fitting a model to a given set of training data, and providing a means for producing predictions for the modeled variables. The challenges to implementing a software quality model were the translation of model input measures from continuous to discrete values, and formulating a structure for the BBN that represented cause-effect relationships within the software life cycle, and was consistent with Kan's three criteria for cause-effect relationships (discussed in Section 2.3). The subsections below details the implementation of the modeling decisions described in Sections 3.2.3 and 3.2.4. For each causal factor of software quality, the specific measures that were used in the model are identified. In addition, the three structural options for the model are presented.

#### *3.2.5.1 Discretization of Model Inputs and Outputs*

Bayesian belief networks are based on relating cause and effects, and are designed to incorporate both objective and subjective inputs to determining an output. As such, it is necessary to represent all variables within the BBN as discrete variables. This section details the way in which the various measurements frameworks outlined in Section 3.2.3 were represented in the model as discrete variables.

Discretization is the process of representing continuous numbers on a discrete scale. It was necessary to convert all measures used as both inputs and outputs to the software quality model to discrete variables in order to accurately represent subjective and objective information simultaneously. It provides a common frame of reference. However, different variables may be represented differently in the model, and may require different granularities. For example, in representing software process, only a "Yes" or "No" indicator of compliance is required, but the Depth of Inheritance Tree software design measure requires a positive integer value. In order to support the range of needs in representing the discrete variables, five different node types were developed.

Table 3.11: Node Types Developed for Software Quality Modeling

| Node Type | No. States | Node Description |
|---|---:|---|
| Compliance Node | 2 | A variable that describes a "yes" or "no" compliance |
| Quarter Node | 4 | Divides a 100% percentage scale into 4 25% intervals |
| Percentage Node | 9 | Divides a 100% percentage scale into 8 12.5% intervals |
| Ratio Node | 9 | Represents a range of ratios between two variables |
| Integer Node | 4 | Represents the positive integer values 1, 2, 3, 4 |

Table 3.11 identifies the five different types of BBN nodes used in this model. Each node type is intended to address a different need in terms of representing continuous variables as discrete variables. The Compliance Node is simply a binary node that indicates a truth value, "Yes" or "No". The compliance node is used frequently in the quality model to identify whether a given software development practice was employed, or indicate whether a quality need was adequately covered in the requirements. The compliance node is appropriate in these cases because they describe whether or not a given activity was performed. For example, one modeled ISO/IEC 15504 practice is that of "Specifying the Software Requirements". The performance of this practice is a binary condition: a project either specified their requirements or not. This example demonstrates how two states sufficiently characterize software processes and practices.

The Quarter Node is used to express the range of values out of four possibilities. For example, in this study, the Quarter Node is used to characterize the proportion of development team skill in a given phase, and for a given skill level. Because the development team size for this study ranges between 1 and 4 software engineers, representing the ratio of developers with the Quarter Node is appropriate, and causes no loss of precision from the original ratio data.

The Percentage Node is used to express a proportion in which the numerator is smaller than the denominator. This node type has a range of [0:1] and is used extensively in this research as the mechanism for expressing normalized values. For example, the ISO/IEC 9126 metrics that measure software quality are typically represented as the ratio of verified requirements to number of requirements. The resulting measure is a proportion of the requirements that were verified, and is normalized by the number of requirements in a given project. To model this proportion as a discrete variable, it was necessary to partition the percentage range into a number of discrete sections, each of which represents a portion of the percentage spectrum. The accuracy of the model depends largely on the granularity of these partitions - a larger number of discrete sections increases the precision of the model with respect to the actual proportion value. Thus, the goal is to maximize the number of discrete states that represent each proportional variable, as system resources will allow (8 states). The Ratio Node is similar to the Percentage Node except that it discretizes the a proportion between two variables where the numerator is larger than the denominator. In this model, this variable measures the Expansion design complexity measures descibed in Table 3.5.

The Integer Node provides a mechanism to introduce a positive integer value into the model. In this case, each integer value corresponds to a unique discrete state. It is used in the model only to represent the Depth of Inheritance Tree (DIT) design complexity measure. Its 4-state range is based on the maximum DIT value in the underlying software engineering data.

The establishment of several node types for use within the BBN allows for a variety of ways that data can be represented as discrete variables within in the model. These node types were all used to characterized the various continuous measurements identified in Section 3.2.3. Table 3.12 identifies the allocation of node types for all variables in the requirements phase. Table 3.13 identifies the allocation of node types for all variables in the design phase. Table 3.14 identifies the allocation of node types for all variables in the implementation phase. Table 3.15 identifies the allocation of node types for all variables in the integration/test subnet. Each table contains four columns. The first column gives the abbreviated name of the node as used in the modeling software. The second column indicates the node type that is used to represent the variable as a discrete in the model. The third column is one of three values, Skill, Process, or Problem that indicates whether this factor is categorized as a driver related to development team skill, software process maturity, or software problem complexity (respectively).

In addition to the inputs to the model, the output of the model is also represented in a discrete form. As mentioned, all of the software product quality metrics are expressed as a Percentage Node. Specifically, these are intended to be the percentage of the requirements associated with that quality attribute which were verified to have been successfully implemented in the software system. While represented discretely in the model, the need exists to convert that discrete value to a continuous value. That is, in order to provide a useful forecast in the case of the software product quality indicators, it becomes necessary to define a function which converts the discrete output of the model to a continuous value of prediction. Because all of the software quality indicators use the same function for transforming continuous values to the discrete Percentage Node format, they may use the same function for transformation from discrete value to continuous value. The formula is outlined in Equation 3.7.

Table 3.12: Allocation of BBN Node Types to the Requirements Phase Variables

| **Node Name** | **Node Type** | **Causal Factor** | **Node Description** |
|---|---|---|---|
| ReqSkill1 | Quarter | Skill | Percentage of development team with Requirements Skill Level 1 |
| ReqSkill2 | Quarter | Skill | Percentage of development team with Requirements Skill Level 2 |
| ReqSkill3 | Quarter | Skill | Percentage of development team with Requirements Skill Level 3 |
| ReqSkill4 | Quarter | Skill | Percentage of development team with Requirements Skill Level 4 |
| SpecifyReqs | Compliance | Process | Did the project specify the requirements? |
| DetEnvImpact | Compliance | Process | Did the project determine the environmental impact? |
| EvalReqsCust | Compliance | Process | Did the project evaluate requirements with the customer? |
| DevValCrit | Compliance | Process | Did the project develop a validation criteria? |
| DevRelStgy | Compliance | Process | Did the project develop a release strategy? |
| UpdateReqs | Compliance | Process | Did the project update requirements from a previous iteration? |
| CommSWReqs | Compliance | Process | Did the project communicate requirement to stakeholders? |
| ReqTraceable | Compliance | Process | Did the project trace requirements back to customer needs? |
| ReqCorrect | Percentage | Problem | Percentage of requirements correctly implemented |
| ReqComplete | Percentage | Problem | Percentage of customer needs covered in requirements |
| ReqTrace | Percentage | Problem | Percentage of requirements traced back to customer needs |
| ReqVolatile | Percentage | Problem | Percentage of requirements changed since baseline |
| QualityNeed | Compliance | Problem | Was there a customer need for this quality attribute? |
| QualityCov | Compliance | Problem | Were the customer needs adequately addressed in the requirements? |

In effect, this transformation takes the midpoint of the percentage range defined in the Percentage Node and linearly scales that midpoint percentage to the original range ($B$). A drawback to this approach is its lack of precision. By taking the midpoint, the model inherently absorbs an error proportional to the size of each partition in the Percentage Node. Currently, there is no other way to prevent the loss of precision, outside of narrowing the range of each partition in the Percentage Node Type. Unfortunately, that range is constrained by the system memory (1 Gigabyte) of the computer used for this research, and is maximized to eight partitions.

Table 3.13: Allocation of BBN Node Types to the Design Phase Variables

| **Node Name** | **Node Type** | **Causal Factor** | **Node Description** |
|---|---|---|---|
| DesSkill1 | Quarter | Skill | Percentage of development team with Design Skill Level 1 |
| DesSkill2 | Quarter | Skill | Percentage of development team with Design Skill Level 2 |
| DesSkill3 | Quarter | Skill | Percentage of development team with Design Skill Level 3 |
| DesSkill4 | Quarter | Skill | Percentage of development team with Design Skill Level 4 |
| DevArchDes | Compliance | Process | Did the project develop an architecture? |
| DevDetailDes | Compliance | Process | Did the project develop a detailed design? |
| DesInterface | Compliance | Process | Did the project design the interfaces? |
| VerDesign | Compliance | Process | Did the project verify the design with the customer? |
| DesTrace | Compliance | Process | Did the project trace the design back to requirements? |
| DIT | Integer | Problem | Depth of the design's inheritance tree |
| DesExpand | Ratio | Problem | Ratio of architectural components to design modules |
| IFFormExpand | Ratio | Problem | Ratio of interfaces to designed interface formats |
| IFProtExpand | Ratio | Problem | Ratio of interfaces to required interface protocols |
| DesCorrect | Percentage | Problem | Percentage of design modules correctly implemented |
| DesComplete | Percentage | Problem | Percentage of customer requirements covered in design |
| DesTrace | Percentage | Problem | Percentage of design traced back to customer needs |
| DesVolatile | Percentage | Problem | Percentage of design modules changed since baseline |

#### *3.2.5.2 Model Structure*

A Bayesian Belief Network represents cause-effect relationships between variables. This section proposes three different BBN model structures as candidates for modeling software quality. Each model structure is described in this section, analyzed and refined in Sections 4.1-4.3, and compared in Section 4.4 in terms of their ability to fit to the training data set, and their ability to accurately predict for unknown values. Recall from Section 2.3 the three criteria for establishing a cause-effect relationship described in Stephen Kan's "Metrics and Models in Software Quality Engineering" [Kan95]: cause precedes effect in time or logic, cause and effect are empirically correlated, and any cause-effect relationship

Table 3.14: Allocation of BBN Node Types to the Implementation Phase Variables

| Node<br>Name | Node<br>Type | Causal<br>Factor | Node<br>Description |
|---|---|---|---|
| ImpSkill1 | Quarter | Skill | Percentage of development team with Implementation Skill Level 1 |
| ImpSkill2 | Quarter | Skill | Percentage of development team with Implementation Skill Level 2 |
| ImpSkill3 | Quarter | Skill | Percentage of development team with Implementation Skill Level 3 |
| ImpSkill4 | Quarter | Skill | Percentage of development team with Implementation Skill Level 4 |
| DevSWUnits | Compliance | Process | Did the project develop the source code? |
| DevUnitVer | Compliance | Process | Did the project develop unit tests? |
| VerSWUnits | Compliance | Process | Did the project verify the source code units? |
| ImpTrace | Compliance | Process | Did the project trace source code back to design? |
| ImpCorrect | Percentage | Problem | Percentage of source code modules correctly implemented |
| ImpComplete | Percentage | Problem | Percentage of design modules covered in implementation |
| ImpTrace | Percentage | Problem | Percentage of source code files traced back to design |
| ImpVolatile | Percentage | Problem | Percentage of source code files changed since baseline |
| QualityPrev | Percentage | Problem | Percentage of source code implementing a quality need |

Table 3.15: Allocation of BBN Node Types to the Integration/Test Phase Variables

| Node<br>Name | Node<br>Type | Causal<br>Factor | Node<br>Description |
|---|---|---|---|
| TstSkill1 | Quarter | Skill | Percentage of development team with Integration/Test Skill Level 1 |
| TstSkill2 | Quarter | Skill | Percentage of development team with Integration/Test Skill Level 2 |
| TstSkill3 | Quarter | Skill | Percentage of development team with Integration/Test Skill Level 3 |
| TstSkill4 | Quarter | Skill | Percentage of development team with Integration/Test Skill Level 4 |
| DevIntStgy | Compliance | Process | Did the project develop an integration strategy? |
| DevRegStgy | Compliance | Process | Did the project develop a regression test strategy? |
| DevIntTsts | Compliance | Process | Did the project develop integration tests? |
| IntSWItems | Compliance | Process | Did the project integrate the software items? |
| TstIntSW | Compliance | Process | Did the project execute the integration tests? |
| RegTstIntSW | Compliance | Process | Did the project perform regression integration tests? |
| DevTstStgy | Compliance | Process | Did the project develop a software test strategy? |
| DevTsts | Compliance | Process | Did the project develop software test procedures? |
| TstSW | Compliance | Process | Did the project test the software system? |
| RegTstSW | Compliance | Process | Did the project regression test the software system? |
| ReqVerCov | Percentage | Problem | Percentage of requirements functionally verified in testing |

### Converting Percentage Node Discrete Values to Continuous Values

Given a percentage value represented by $P = \frac{A}{B}$, where P is categorized into a Percentage Node discrete format consisting of $N$ state ranges (as described in Table 3.11), and a corresponding state number $S > 1$ which has been output by the quality model. A continuous value V is determined by:

$$V = B * [(S - 2) * \frac{1}{N}] + (\frac{1}{N} * \frac{1}{2}) \tag{3.7}$$

is logical. This research defines three model structure that satisfy these three criteria. This section describes each candidate model structure in terms of the logic of the cause-effect relationships. Once the intuitive structure is established, a statistical analysis of each model's structure is performed as part of the analysis of the results in Chapter 4. The resultant streamlined model structures are used to calculate and compare model results.

The model structures proposed in this paper are based largely on three premises for software product quality:

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 question becomes how to relate those factors to software quality. Using the measures described in Section 3.2.5.1, it is logical that each phase of the software life cycle carries with it a particular set of needed skills, unique processes, and brand of complexity. That is, each phase of the software life cycle is directly affected by these high level causal factors, and contributes in turn to the ultimate quality of the finished software product. Also, each life cycle phase produces an artifact that is the basis for quality in subsequent phases. Consider the cause-effect, or fishbone, diagram shown in Figure 3.3 that details the internal relationships between causal factors. Quality is affected by the correctness and completeness of the activities and artifacts at each phase of the development life cycle. Measures of completeness and correctness are driven by the development team's skill/experience level, the process maturity or infrastructure, and the problem complexity for each phase.

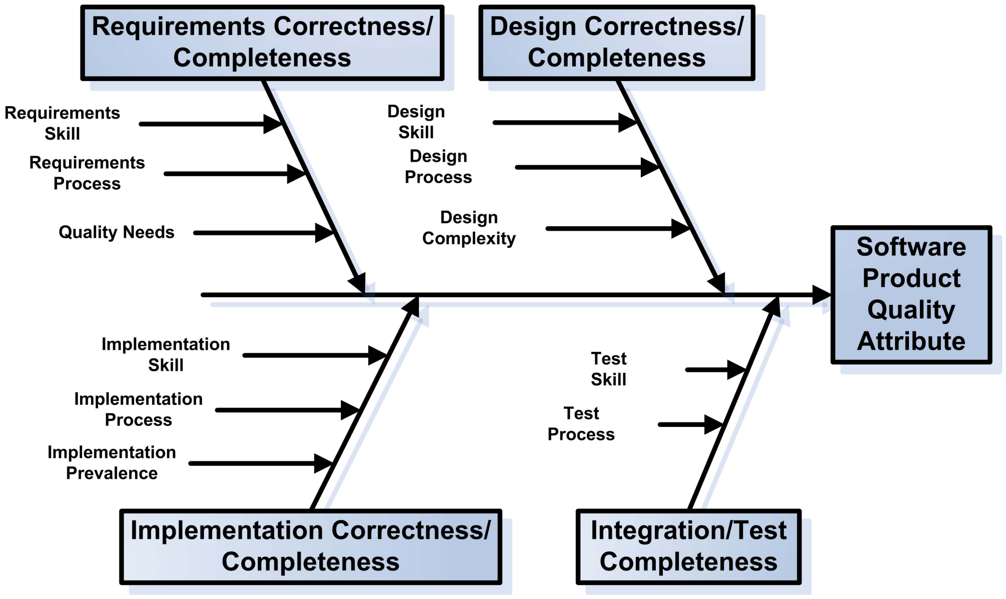
Requirements Correctness/ Completeness
Design Correctness/ Completeness
Requirements Skill
Requirements Process
Quality Needs
Design Skill
Design Process
Design Complexity
Software Product Quality Attribute
Implementation Skill
Implementation Process
Implementation Prevalence
Test Skill
Test Process
Implementation Correctness/ Completeness
Integration/Test Completeness

Figure 3.3: Software Quality Cause Effect Diagram

The first model structure proposed is a three-tiered structure that directly implements the cause-effect diagram shown in Figure 3.3. The structure of the model, and the delineation of the tiers is shown more clearly in Figure 3.4. The input tier contains the model inputs, including the measures of development team skill/experience, process maturity, and problem complexity for each life cycle phase. The intermediary tier contains correctness and completeness variables for each life cycle phase. All model inputs indigenous to a life cycle phase (see Section 3.2.5.1) are linked in the model structure to the correctness and completeness variables for that phase. Finally, the output tier contains the software quality variable itself, and is a product of the correctness and completeness measures from each life cycle phase. This structure will be referred to hereafter as the Intuitive Model. It is based on the three premises identified above, and uses measures of correctness and completeness of life cycle artifacts as the intermediary indicators of software quality. The Intuitive Model is further refined through an analysis of the model variables Section 4.1 to eliminate invariant inputs, and cases of multicollinearity.

A weakness in the Intuitive Model is that it is not rigorous in its formulation. That is, while the structure is reasonable from a logical standpoint, there is no quantifiable basis for the relationships. To that end, the second model structure is proposed, which is labeled the Refined Model. The Refined Model retains the three tier structure of the Intuitive Model, but has been subject to a statistical analysis in which cause-effect relationships are retained only if they can be correlated as described in Section 3.3.2.2. The structure of the Refined Model is discussed as part of the analysis performed in Section 4.2.

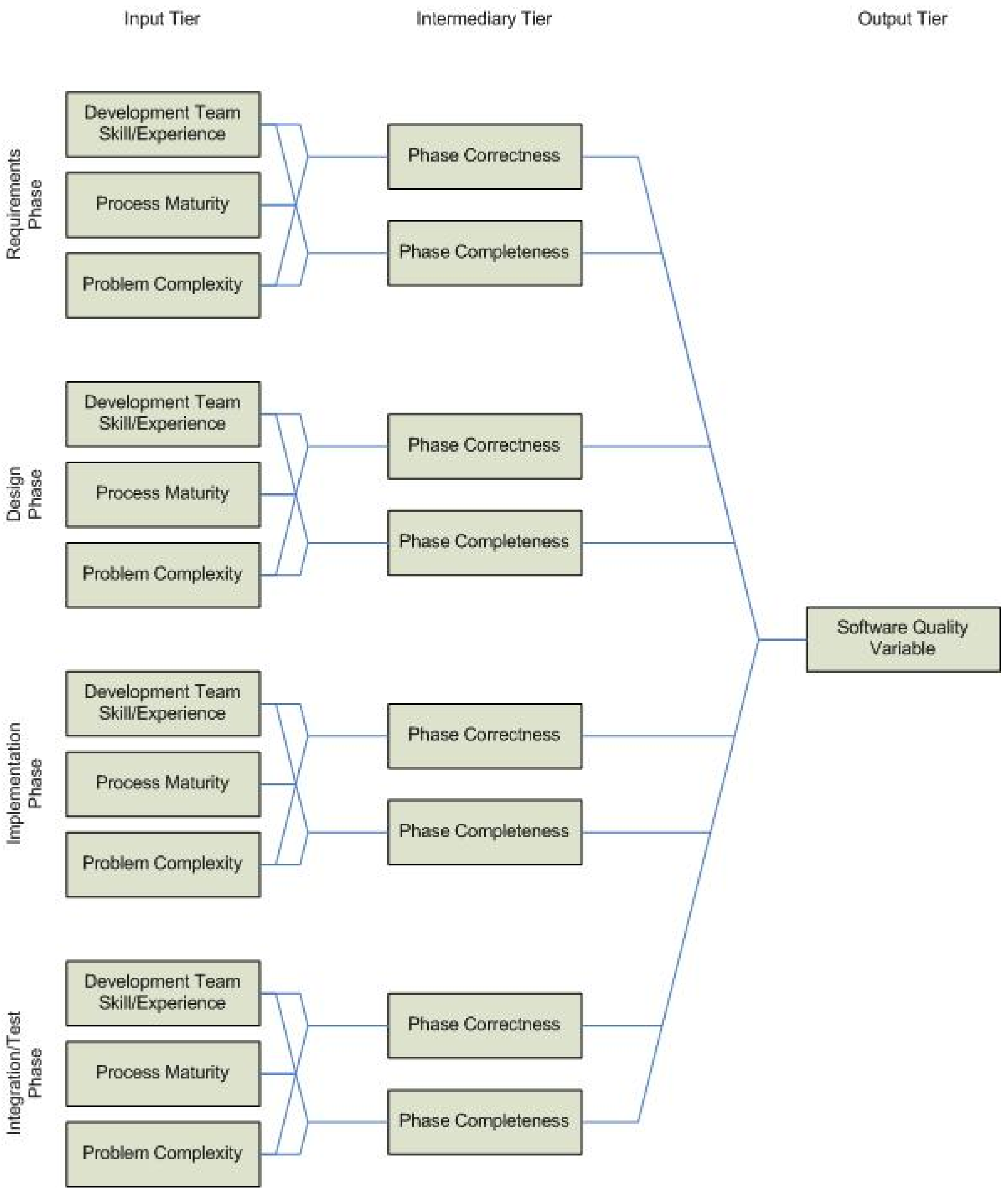

Figure 3.4: Intuitive and Refined Model Structures

The final proposed structure takes a much more traditional approach to determining model structure. Model inputs are analyzed to determine if they are correlated with software quality variables. Any correlations established using the procedure described in Section 3.3.2.2 indicate a link in the Bayesian Belief Net structure. This model structure has been labeled the Direct Effects model as it simply related inputs to outputs in a two tier model structure (see Figure 3.5). The Direct Effects Model considers the full set of inputs and does not discriminate between life cycle phases. That is, all model input variables are tested for valid correlations with each quality output, and without regard to the life cycle phase to which the measure was categorized.

#### *3.2.5.3 Tool Selection*

This section identifies the software packages that were used to develop, and execute the software quality model in order to make forecasts of software quality. The model was to be developed as a software package that would be able to read in a data file, implement the BBN programmatically, forecast software quality using conditional probabilities calculated from prior data sets, and automate any validation of the forecasts. This research required decisions in terms of the Bayesian Belief Network libraries that would be used, the language in which the model and validation software would be written, and the development environment that would be used.

A commercial tool was selected for the Bayesian Belief Network libraries. The Netica ®package, by Norsys Software Corporation ®[Cor06], provided the basic software classes that allowed for the construction of the Bayesian Belief Network, the initial calculation of conditional probabilities based on existing data sets, and the execution of the model in order to make a inferences about the various software product quality measures. The

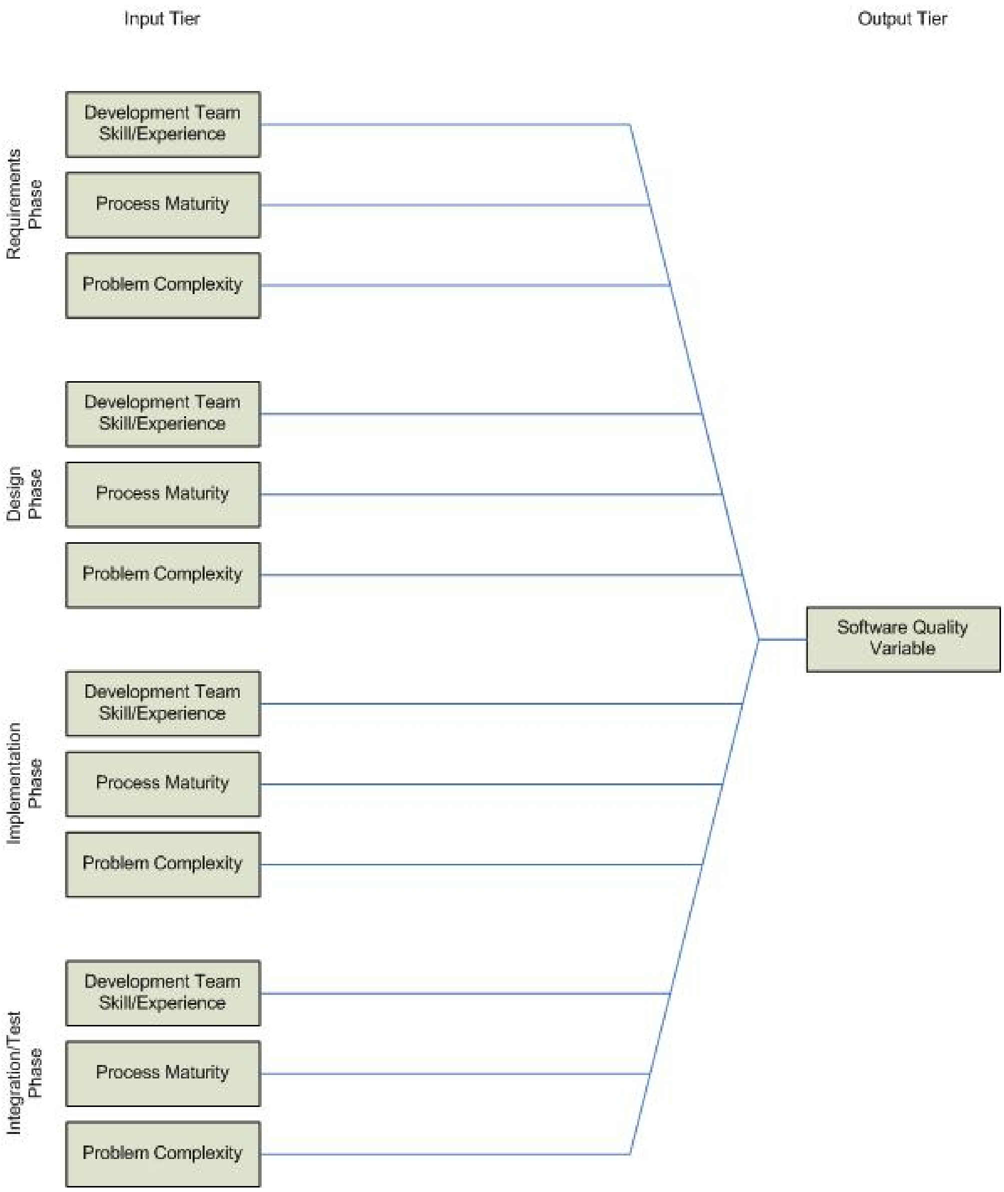

Figure 3.5: Direct Effects Model Structure

Netica package provided facilities for creating BBN nodes and arcs, and establishment of the conditional probability tables associated with each node.

All of the software developed for this research was done in the Java programming language, version 1.5. Java was selected for its ease of programming, and also for its compatibility with the Netica ®libraries. The custom code developed served to read in the data files, define the node types to be used, programmatically establish the BBN structure, and programmatically provide forecasts and validation of those forecasts. All development was done using the Eclipse ®integrated development environment. Eclipse ®[Fou06] is an open-source product that is freely available.

## 3.3 Validation Approach

Validating the proposed software quality model involves collecting a set of software engineering data, and using that data to confirm the ability of the model to characterize the data set, and the ability of the model to make predictions for unknown data sets. This section describes the approach to these activities. It begins by outlining the methods used to acquire the software engineering data. The software engineering data set is a combination of process, product, and skill data from small-scale industry and graduate student software development projects. This section follows with a description of the statistical tests and formulas used quantify accuracy of fit and predictive validity. Finally, the approach to comparing the developed software quality model to quality models using other modeling mechanisms is outlined.

The purpose of this research is to provide a reliable model for predicting software quality in a given development project. The model attempts to establish the causal influences of personnel skill, problem scope, and process maturity as driving factors in the ultimate quality of the software product. The following list of questions will be answered by through validation of the model:

1. Does the software quality model provide an accurate assessment of software product quality?
2. Does the software quality model accurately predict software product quality while the software is under development?

Initially, a determination is made as to the optimum model structure for each software quality variable. Section 3.2 outlined three possible candidates for a BBN structure that are suitable for a given software quality variable. The Intuitive Model structure is the initially proposed, logically inferred structure that has not been adjusted by any mathematical corroborations of cause-effect relationships. The Refined Model retains the three-tiered structure of the Intuitive Model, but has been streamlined to only include those cause-effect relationships that have been statistically correlated. The Direct Effects Model structure abandons the idea of a three-tier BBN structure, and simplifies the BBN to an association of software quality outputs to correlated model inputs. The first step in validating the model is to select, based on measures of Accuracy of Fit and Predictive Validity proposed below, the most appropriate BBN structure for each software quality variable. Once a Bayesian model structure is selected, the results for each software quality attribute are compared to competing approaches to modeling software quality.

The techniques for validation build upon a previous study in which a smaller scale software quality model was validated [BSB05]. Two factors are considered when validating a software quality model: Accuracy of Fit as expressed through verifying the equality of means and variances between predicted and actual results, and a measure of the Predictive Validity or accuracy of the model's forecasts. Since the cost to correct software increases exponentially as the software life cycle progresses [Boe81], it is useful to project software quality as early as possible in order to minimize rework and maximize the potential cost savings. However, without some measurable insight into the architecture of a software system, it is difficult to lend credence to any forecasts made prior to the formalization of the system's design. The conclusion of the design phase was selected as the optimum time in the development life cycle to make forecasts of software quality. At this point, there is enough information about the complexity of the software problem and its design solution to give credibility to the projected quality of the final product. By the same token, it is early enough in the life cycle to make improvements with minimal impact to the project's schedule and budget.

Figure 3.6 shows the validation process. A ”leave one out” cross-validation approach was to taken to evaluate the model's ability to provide reliable forecasts of software quality. In this approach, the model is trained using all but one of the available projects. The remaining project's life cycle data is input into the model in order to make predictions about software quality, and verify the accuracy of those predictions versus the actual values. As shown in the diagram, the model's software quality predictions for the remaining project are acquired at the conclusion of the design phase. This process was repeated for each of the projects in

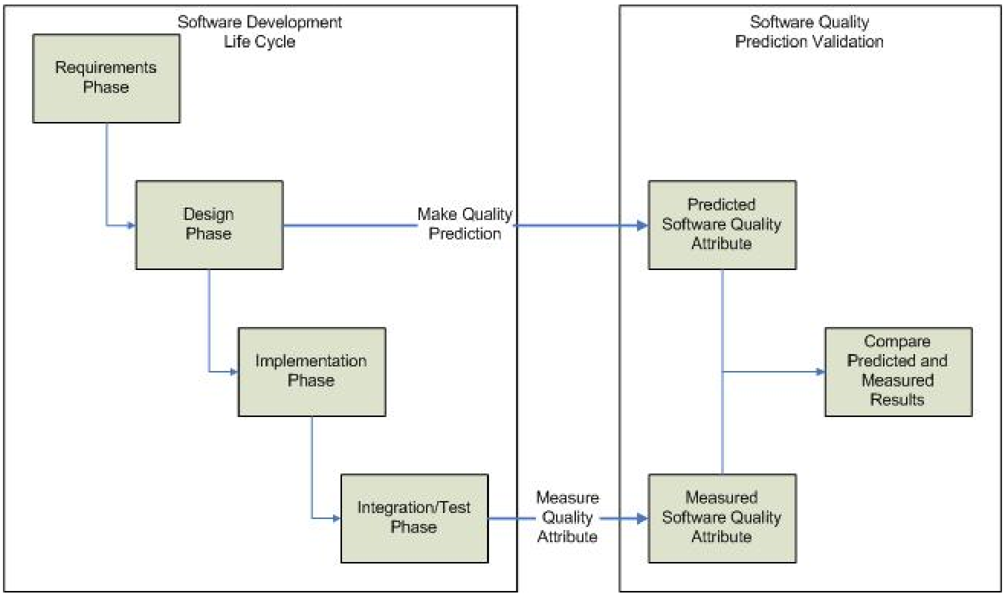

Figure 3.6: Process for Validation of Software Quality Model Predictions

the data set. The resultant data is a collection of predicted and actual values for each of the software quality measures being validated.

### *3.3.1 Acquisition of Software Engineering Data*

This section details the approach to acquiring software engineering data for training and validating the developed software quality model. Model validation will be through analysis of data collected from both student and industry software development efforts. In the case of the student software development projects, personnel and process for each project was varied to create a broader range of model input variables in order to provide more complete coverage of the input state space. For the industry projects, a post-deployment analysis of the project's artifacts provided the data necessary to serve in this study's data set. This section describes the set of data collected for this study, and the approach to data collection.

#### *3.3.1.1 Data Collection Procedures*

This section details the approach to the acquisition of software personnel, process, and product measurements for use in the developed software quality model. The purpose of this data is to establish the initial conditional probabilities, or training data, associated with the relationships defined in the model. These conditional probabilities establish the likelihood of the paths taken through the model structure.

Table 3.16: Data Collection Activities Relative to the Software Life Cycle

| **Life Cycle Event** | **Data Collection Activities** |
|---|---|
| Project Planning | Software Development Skill Questionnaire administered<br>Overview of Metrics to be collected |
| Software Specification Baseline | Collect Requirements Phase metrics |
| Software Design Baseline | Collect Design Phase metrics |
| Source Code Complete | Collect Implementation Phase metrics |
| Integration/Test Complete | Collect Software Product Quality measures |

The data collection period for each of the participating software engineering projects was the complete life cycle of their development. Table 3.3.1.1 provides a schedule of data collection activities that were performed and the points in the development life cycle where data was collected. Initially, the developers for each project were required to fill out the skill questionnaire (see Appendix A) in order to assess the capabilities of the development team.

Also, the development team was briefed on the types of metrics that would be collected, and the importance of providing accurate measures. At the conclusion of each phase of the life cycle, the metrics for that phase would be collected. After the developed product had been tested as a software system, the final set of metrics were collected, and the final values of software product quality were calculated.

#### *3.3.1.2 Description of Acquired Software Engineering Data*

The sets of data used to train and validate this software quality model were acquired from 28 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 35 projects selected to participate in this research, 28 were actually used because of their willingness to track life cycle measures completely and correctly. The raw data collected and used in this study to train and validate the BBN is presented in Appendix C.

### *3.3.2 Methods of Data Analysis*

This section details the various methods used to analyze the acquired software engineering data, and validate the proposed software quality model. As part of validating cause-effect relationships in the model structure, the approach to calculating and verifying statistical correlations is described. In addition, the methods used to quantify the proposed model's Accuracy of Fit and Predictive Validity are presented.

#### *3.3.2.1 Calculating Mean and Variance*

The mean and variance are the most basic elements of statistical analysis. Consider a set of $n$ values, $y_1, y_2, y_3, \ldots, y_n$. The mean ($\mu$) is the average of those values and is calculated using Equation 3.8.

$$\mu = \frac{1}{n} \sum_{i=1,N} y_i \tag{3.8}$$

The variance ($\sigma^2$) quantifies the sum of squared deviations of each value from the mean, and is calculated using Equation 3.9.

$$\sigma^2 = \frac{1}{n} \sum_{i=1,N} (y_i - \mu)^2 \tag{3.9}$$

An analysis of the mean and variance of model variables is performed in Section 4.1, and is useful in identifying high-level trends in the model data, and identifying invariant model inputs.

#### *3.3.2.2 Establishing Statistical Correlation*

Much of the model structure analysis is accomplished by calculating and interpreting the correlation of input variables to both software quality measures and to downstream nodes within the model structure. Correlation, typically denoted with the variable $r$, is a measure of the strength of a linear relationship between two variables. Correlation values were calculated using Equation 3.10 in which two variables, $x$ and $y$, are evaluated over $N$ samples. Determining the significance of any correlations is a necessary step in establishing a cause-effect relationship between two variables [Kan95]. Since the this software quality model represents cause-effect relationships, establishing and analyzing the linear correlation of variables is appropriate.

$$r = \frac{\sum_{i=1,N} (x_i - \overline{x}) (y_i - \overline{y})}{\sqrt{\sum_{i=1,N} (x_i - \overline{x})^2 \sum_{i=1,N} (y_i - \overline{y})^2}} \tag{3.10}$$

The calculation of correlation between two variables is an important step, but not sufficient in establishing a cause-effect relationship. It is necessary to verify with quantifiable certainty that the correlation is not merely a random occurrence. A Hypothesis Test is conducted in conjunction with the correlation calculation in order to determine whether the population correlation coefficient ($\rho$) is sufficiently characterized by the sample correlation coefficient $r$. The Hypothesis Test is summarized in Table 3.17 [MS95]. The null hypothesis ($H_0$) asserts that variable $x$ contributes no information for predicting $y$, and the alternative hypothesis ($H_a$) asserts that the two variables are correlated. The null hypothesis is rejected if the Test Statistic ($t$), which is calculated using the sample size ($n$) and the sample correlation coefficient ($r$), is larger in magnitude than the Student's T critical value for a confidence value of 90%. A rejection of the null hypothesis is interpretted as a determination, with a confidence of 90%, that the relationship between variables $x$ and $y$ is an actual linear correlation, and not a random occurrence between the two data sets. In other words, it provides a quantifiable level of confidence in the correlation values determined.

Table 3.17: Hypothesis Test for Verification of Linear Correlation

$H_0$: $\rho = 0$
$H_a$: $\rho \neq 0$

Reject $H_0$ if:

$$|t| > t_{\alpha/2}$$

where,

$$t = TestStatistic = \frac{r * \sqrt{n-2}}{\sqrt{1-r^2}}$$

and,

| | |
|---|---|
| r | = the sample coefficient of correlation |
| n | = the total number of samples |
| $t_{\alpha/2}$ | = Student's T Critical Value for confidence (1 - $\alpha$)100% |

#### *3.3.2.3 Determining Accuracy of Fit*

The Accuracy of Fit of a model is a quantitative determination of how well a model represents the data upon which it is based. In other words, it is a measure that indicates the correctness of the model with respect to the data that was used to construct the model. Accuracy of Fit is determined through an analysis of the Equality of Means and the Equality of Variances between the modeled values of software quality and the actual values.

Verifying the Equality of the Means between the predicted and actual values of software quality gives the model credibility in terms of its ability to accurately represent the underlying data. 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 ($H_0$) is that the means are equal, and the alternative hypothesis ($H_a$) is that they are different. The decision rule and associated calculations for the Hypothesis Test is shown in Table 3.18[MS95].

The Equality of Means Hypothesis Test (see Table 3.18) quantifies the confidence that the mean value of two populations are equivalent. By comparing the Equality of Means between actual software quality values and modeled software quality values, the ability of the model to accurately characterize the underlying data set is revealed. If the Test statistic ($t$) for the given quality measure is less than the t-distribution value ($t_{n-\nu,\alpha/2}$) for that quality measure, then the null hypothesis must be accepted, the means are determined to be equivalent, and the model is said to provide an accurate fit for the underlying data. For this

Table 3.18: Hypothesis Test for Determining Equality of Means

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

Reject $H_0$ if:

$$|t| > t_{n-\nu,\alpha/2}$$

where,

$$t = TestStatistic = \frac{\mu_{actual} - \mu_{modeled}}{\sqrt{msE * (2/n)}}$$

and,

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

study, a confidence of $\alpha = 0.9$, or 90% was used for all Equality of Means calculations. This means that there is 90% confidence that all Equality of Means determinations are correct.

The approach to calculating the Equality of Variances is to use a textbook rule of thumb test recommended in [DV99], and described in Table 3.19. In comparing the modeled data to the actual data, if the ratio of the maximum variance value to the minimum variance value must be less than three to consider the variances equivalent. The application of this rule of thumb is appropriate in that it bounds the relationship of the variances. The goal for the Equality of Variances is not to get a quantifiable confidence on the accuracy of the modeled data (that is accomplished through the Equality of Means test), but to get a discrete indication that the variance of the modeled data is on the order of the variance of the actual data.

Table 3.19: Test for Determining Equality of Variances

Variances $\sigma_1^2$ and $\sigma_2^2$ are equivalent if:

$$\frac{\sigma_{max}^2}{\sigma_{min}^2} < 3$$

where,

$\sigma_{max}^2$ = the maximum value between $\sigma_1^2$ and $\sigma_2^2$
$\sigma_{min}^2$ = the minimum value between $\sigma_1^2$ and $\sigma_2^2$

#### *3.3.2.4 Determining Predictive Validity*

The Predictive Validity is a measure of how accurately the model predicts a variable using an unknown data set as input. It is a quantitative way of determining how well a given model characterizes an unknown. Predictive Validity is measured using the Average Relative Error (ARE). ARE describes the deviation of the actual data from the modeled data, and is defined in Equation 3.11 [KBR92].

$$ARE = \frac{1}{N} * \sum_{i=1,N} \left| \frac{(\overline{y}_i - y_i)}{y_i} \right| \tag{3.11}$$

In Equation 3.11, $N$ is the number of data elements, $\overline{y}_i$ is the modeled value of the data, and $y_i$ is the actual data value. By definition, the lower the value of ARE, the more closely the model approximates the actual data. Because the ARE is a measure of deviation of estimated values from actual values, it is appropriate to use in determining how closely a model represents any data set. The ARE in determining Predictive Validity provides assurance that the models chosen are reliable in making predictions about unknown data. ARE has already been confirmed as a significant measure for the Predictive Validity of software quality models [BS03b] [SYT85], and an ARE of 0.25 or less is considered acceptable [CDS86] to be a useful value of prediction.

### *3.3.3 Competing Models Description*

This section describes the competing models with which the Bayesian Network will be compared. In Chapter 2, an evolution of software quality modeling was described that culminated in the identification of Complex Adaptive Systems, and particularly Bayesian Belief Networks, as the method of choice for modeling software quality. The validation of the selected Bayesian network structure will involve a comparison to competing approaches for modeling software quality. This section describes the Least Squares Regression model and the Neural Network model. Each of these will be compared to the Bayesian model in terms of both Accuracy of Fit and Predictive Validity.

#### *3.3.3.1 Least Squares Regression*

Least Squares Regression is a technique for fitting a model to an underlying data set. It involves calculating the set of weights that correspond to each model input so as to minimize the sum of squared error term for the output. Consider a set of model inputs denoted by the variable $x$ and a model output denoted by the variable $y$ as shown in Equation 3.12. The $y$ variable can be expressed in terms of the inputs $(x)$ and a weight for each input $(\beta)$ that provides the best fit for the set of data, and minimizes the error term $(\epsilon)$.

$$y = \beta_{constant} + x_1\beta_1 + x_2\beta_2 + x_3\beta_3 + \ldots + x_m\beta_m + \epsilon \tag{3.12}$$

However, in order to fit to a set of data, each instance of this equation must be considered. Table 3.20 is an example data table of how Equation 3.12 might be denoted for n data sets and m inputs. The resultant array of equations can be represented more succinctly as matrices, as in Equation 3.13.

$$Y = X\beta + \epsilon \tag{3.13}$$

Thus, the aim of the Least Squares Regression model is to determine the matrix of weights $(\beta)$ that will minimize the matrix of error terms $(\epsilon)$ when using the inputs $(X)$ to model the outputs $(Y)$. With the set of input and output data available, the determination of weights simply becomes a matter of matrix algebra. Solving for the weights $(\beta)$ in Equation 3.13, Equation 3.14 can be used to solve for the optimum set of inputs weights.

$$\beta = (X^TX)^{-1}X^TY \tag{3.14}$$

Table 3.20: Least Squares Example Data

| Data | Output | Inputs | | | | | Random |
|---|---|---|---|---|---|---|---|
| Set | y value | $x_1$ | $x_2$ | $x_3$ | $\ldots$ | $x_m$ | Error |
| 1 | $y_1$ | $x_{1,1}$ | $x_{1,2}$ | $x_{1,3}$ | $\ldots$ | $x_{1,m}$ | $\epsilon_1$ |
| 2 | $y_2$ | $x_{2,1}$ | $x_{2,2}$ | $x_{2,3}$ | $\ldots$ | $x_{2,m}$ | $\epsilon_2$ |
| 3 | $y_3$ | $x_{3,1}$ | $x_{3,2}$ | $x_{3,3}$ | $\ldots$ | $x_{3,m}$ | $\epsilon_3$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| n | $y_n$ | $x_{n,1}$ | $x_{n,2}$ | $x_{n,3}$ | $\ldots$ | $x_{n,m}$ | $\epsilon_n$ |

In this study the Least Squares Regression was implemented as custom developed software. The set of inputs used for each software quality variables were those causal factors that were verified to be correlated to the quality attribute in the Direct Effects analysis (see Section 4.3). The procedure for prediction was to calculate the optimum set of weights that would minimize the error associated each software quality variable. Once the weights were determined, they were used to predict software quality using an unfamiliar set of input values. The intent of using Least Squares Regression as a basis for comparison is to demonstrate that the Bayesian models are an improvement over a past approach.

#### *3.3.3.2 Neural Networks*

Neural Networks are similar to Bayesian Belief Networks in that they are also an adaptive modeling method, use prior data sets to train the model, and can be represented graphically as a network of nodes with arcs depicting relationships. Neural Networks are included in this research as a basis for comparison in order to demonstrate how Bayesian models outperform other modern approaches to modeling software quality. Similar to the Least Squares Regression, Neural Networks aim to fit to a set of training data by weighting each input based on prior examples of input and output values.

Figure 3.7 shows the structure of a simple Neural Network. It is comprised of input nodes (or neurons) which take in data values for both learning and modeling, and output nodes which produce modeled results. A unique aspect of the Neural Network is that only the input and output nodes have explicit meaning. There is an intermediary set of nodes, called hidden nodes that have no real-world meaning, but are the mechanism by which the network can be trained to produce an optimum set of weights for the inputs. Each layer of nodes is completely connected to other layers through a network of associations called a synapse. Neural Networks are trained by inputting values for the model inputs at the input layer, and comparing the results at the output with actual data values. Through a repetition of training the model with the set of known data, the weight of each synapse connection is adjusted until the output most accurately models the actual output values.

Similar to the approach taken in Least Squares Regression, this study constructed a Neural Network that emulated the Direct Effects Bayesian Model. Model inputs were directly associated to each software quality output (although through a set of hidden nodes). The ”leave one out” cross-validation approach applied to the developed Neural Network: the network was trained using all but one data set, and that data set was used to predict the software quality attributes. The Neural Network used in this research was implemented using

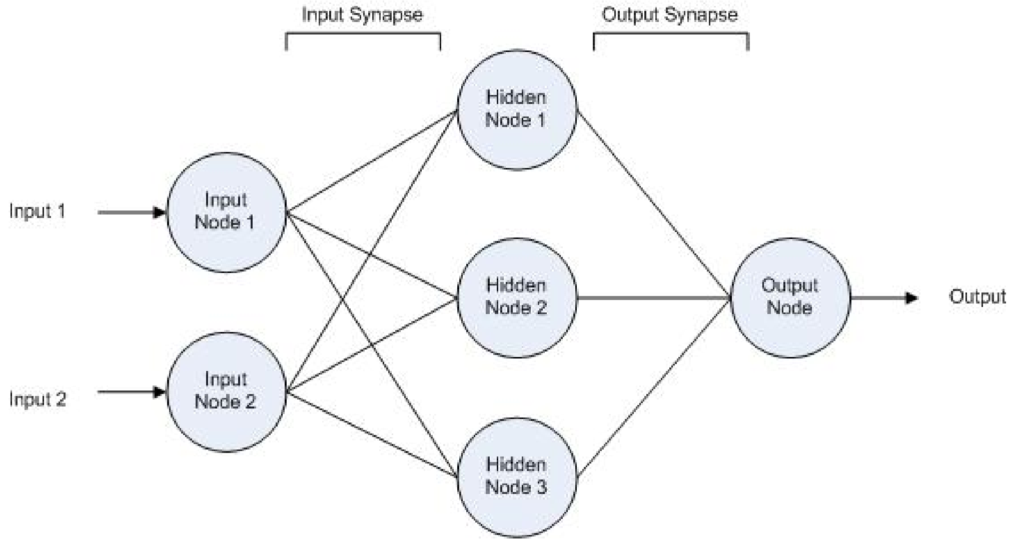

Figure 3.7: Example Neural Network

the Java Object-Oriented Neural Engine (JOONE) Artificial Neural Network framework [Mar04].

# CHAPTER 4

# RESULTS

This chapter reports the results of the acquisition of the software engineering data, and the use of that data to model software quality. Specifically, the proposed software quality model is statistically analyzed, and the model's structure is refined to include only the most significant contributing factors. In addition, the results of the model's validation are presented, and those results are compared to the software quality predictions of competing models.

## 4.1 Analysis of Software Quality Variables

This research attempts to incorporate many different types of software engineering measures into a comprehensive software quality model that assesses and predicts quality in a software development effort. The measures and model structure proposed in Chapter 3 are comprehensive in terms of the number of possible causal factors and software quality variables that are significant in the model. Using the acquired software engineering data as the foundation for analysis, it is appropriate to further scrutinize the proposed set of software quality measures that comprise the model. The purpose of this analysis is to identify those variables that are not significant in modeling software quality, and therefore may be eliminated from the model structure. The inputs and intermediary nodes in the model are analyzed in terms of their mean and variance, as these values give insight into the underlying data, and reveal invariant variables that add no value in terms of a cause-effect model. The modeled software quality variables are analyzed in terms of their sample sizes, with the goal of determining whether that size is adequate make a statistical inference.

### *4.1.1 Mean and Variance Analysis*

The mean and variance of the causal factors that comprise the proposed software quality model can provide insight into the nature of the underlying software engineering data. This section provides an analysis of the mean and variance for each input and intermediary node in the software quality model structure. The intent of this analysis is to identify patterns across the causal factors with respect to the collected software project data. In addition, this analysis provides an opportunity to refine the complexity of the model structure and exclude from the model those variables that are invariant across the data set. Because Bayesian Belief Networks are based on the conditional probabilities associated with prior data sets, a quality factor that does not vary in value cannot provide any discrimination in terms of its effect on software quality variables being modeled. It is therefore appropriate to remove from the model those factors that do not vary in value and thus are not useful in terms of either assessment or prediction. The mean and variance of the input and intermediary nodes modeled in each life cycle phase are discussed in detail in the subsections below. As the analysis of mean and variance provides a broad but not a conclusive sense of the underlying data, this section references other areas of this document where some of the wide-scale implications of the data are explored in more detail.

#### *4.1.1.1 Requirements Phase Variables*

The mean and variance of the Requirements phase variables are presented in Table 4.1. The personnel that comprised the development teams used to acquire software engineering data were primarily intermediate-level software requirements developers. That is, software development teams on the whole were staffed by engineers with some prior experience in the elicitation and documentation of requirements. Of particular note is the lack of requirements "experts", or skill level 4 engineers. The presence of software requirements experts and their effect on the software quality are discussed in more detail in Section 4.3.7.

From a process perspective, almost all projects in the software engineering data set specified software requirements, and developed some form of validation criteria for acceptance of the software product. The remainder of the requirements phase processes were largely not performed across the set of projects. The most variant requirements processes are the practices surrounding an evaluation or review of the developed requirements versus either the

Table 4.1: Mean and Variance of Requirements Phase Variables

| Model Variable | Samples | Mean | Variance |
|---|---|---|---|
| Requirements Level 1 | 28 | 0.1696 | 0.1014 |
| Requirements Level 2 | 28 | 0.5208 | 0.1531 |
| Requirements Level 3 | 28 | 0.2560 | 0.1121 |
| Requirements Level 4 | 28 | 0.0536 | 0.0105 |
| Specify Software Requirements | 28 | 0.9286 | 0.0663 |
| Evaluate Requirements With Customer | 28 | 0.5714 | 0.2449 |
| Update Requirements | 28 | 0.1786 | 0.1467 |
| Communicate Software Requirements | 28 | 0.2143 | 0.1684 |
| Determine Environmental Impact | 28 | 0.2143 | 0.1684 |
| Develop Release Strategy | 28 | 0.2143 | 0.1684 |
| Develop Validation Criteria | 28 | 0.7857 | 0.1684 |
| Evaluate Software Requirements | 28 | 0.3571 | 0.2296 |
| Requirements Correctness | 28 | 0.9777 | 0.0044 |
| Requirements Completeness | 28 | 0.8018 | 0.0565 |

customer or traceability to the identified customer needs. Intuitively, these two process areas would seem to have a direct effect on the correctness and completeness of the requirements. These will be analyzed further in Section 4.2.1.1.

The intermediary nodes of completeness and correctness reveal that the projects on the whole were thorough in their requirements development efforts. Both completeness and correctness mean values were high, and with little variance, which indicates a consistency across the various projects. This consistency can be detrimental in terms of the requirements model structure. A lack of variance in these intermediary nodes has the potential to mask or block the effects of the upstream parameters that may have otherwise had a stronger influence on the various software quality variables. The potential for altering the model structure to remove requirements correctness and completeness as intermediary nodes is discussed in Section 4.2.2.1.

#### *4.1.1.2 Design Phase Variables*

Table 4.2 contains the mean and variance values for the input and intermediary nodes that comprise the Design life cycle phase. The distribution of skill is similar to that of the requirements phase: a majority of intermediate software designers with some beginners and few experts. The small proportion and variance associated with design experts calls into question the significance of the influence of experienced designers on the quality factors represented in the model.

Table 4.2: Mean and Variance of Design Phase Variables

| Model Variable | Samples | Mean | Variance |
|---|---|---|---|
| Design Level 1 | 28 | 0.1577 | 0.0855 |
| Design Level 2 | 28 | 0.5506 | 0.1599 |
| Design Level 3 | 28 | 0.2262 | 0.1135 |
| Design Level 4 | 28 | 0.0655 | 0.0131 |
| Develop Software Architecture | 28 | 0.8571 | 0.1224 |
| Develop Detailed Design | 28 | 0.8571 | 0.1224 |
| Verify Design | 28 | 0.5714 | 0.2449 |
| Design Interfaces | 28 | 0.8571 | 0.1224 |
| Design Traceability | 28 | 0.3571 | 0.2296 |
| Depth of Inheritance Tree | 28 | 1.6071 | 1.2385 |
| Design Expansion | 28 | 0.7962 | 0.8486 |
| Interface Format Expansion | 19 | 0.9048 | 0.4741 |
| Interface Protocol Expansion | 18 | 1.1944 | 0.3650 |
| Requirements Volatility | 28 | 0.1511 | 0.0677 |
| Design Correctness | 14 | 0.9121 | 0.0219 |
| Design Completeness | 28 | 0.9856 | 0.0025 |

Those processes associated with documenting the software design were largely practiced across the set of subject projects. The lesser practiced of the design processes, namely the verification of design and traceability were the most variant. In terms of design complexity, there was significant variance for all of the measures used. The increases the viability of these metrics as discriminators for software quality.

As with the requirements phase, correctness and completeness of the software design were both consistent across projects, and largely invariant. Thus, it is likely that the presence of these nodes as intermediaries in the Bayesian network may mask significant variations in upstream parameters. Further analysis was performed (see Section 4.2.2.2) to assess the necessity of these nodes in the network structure.

#### *4.1.1.3 Implementation Phase Variables*

Table 4.3 contains the mean and variance values for the input and intermediary nodes in the Implementation life cycle phase. On the average, skill level in the implementation phase was higher than that of the requirements and design phase. Nearly 90% of the developers had some prior experience as authors of software. Also, the presence of experts was stronger in the implementation phase, averaging nearly 1 in 5. The higher implementation skill is not surprising since implementation is typically the focus when learning software. Beginning software engineers start to learn a programming language and/or various methods of software implementation long before they are introduced to the engineering of software through sound requirements elicitation and design development. Thus, the quantity of experience with the implementation aspect of software development is higher.

Table 4.3: Mean and Variance of Implementation Phase Variables

| Model Variable | Samples | Mean | Variance |
|---|---|---|---|
| Implementation Level 1 | 28 | 0.1190 | 0.0969 |
| Implementation Level 2 | 28 | 0.4137 | 0.1872 |
| Implementation Level 3 | 28 | 0.2827 | 0.1152 |
| Implementation Level 4 | 28 | 0.1845 | 0.0944 |
| Develop Software Units | 28 | 1.0000 | 0.0000 |
| Develop Unit Tests | 28 | 0.2143 | 0.1684 |
| Verify Software Units | 28 | 0.4643 | 0.2487 |
| Software Unit Traceability | 28 | 0.3571 | 0.2296 |
| Design Volatility | 28 | 0.1689 | 0.0849 |
| Implementation Correctness | 28 | 0.8551 | 0.0378 |
| Implementation Completeness | 28 | 1.0000 | 0.0000 |

Two of the model inputs in the implementation phase can be eliminated due to a lack of variance. The process step that includes the development of software units was practiced by all projects. Since there is no variance in this process step across the set of project data, this particular data input becomes useless within a Bayesian Belief Network. The BBN requires variation in the a variable in order for values across input states to change in the conditional probability tables. In addition, the intermediary node Implementation Completeness, which measures the proportion of the designed software modules that were captured in the implementation, was invariant over the set of software engineering data. These two variables will be removed from the model.

#### *4.1.1.4 Integration/Test Phase Variables*

The distribution of skill in the Integration/Test Phase is more uniform than in the other life cycle phases. That is, each of the skill levels is more equally represented for each development project in the data set. Of particular note is the strong presence of testing experts. The effect of this on the various aspects of software quality are analyzed in Section 4.3.

Several of the process inputs associated with the Integration and Test phase are not useful in the model structure because they are invariant across the project data set. For example, all of the software development projects tested their system, and so this process area can add no value as a discriminator for software quality. As with the other phase completeness measures, Test Completeness will need further analysis (performed in Section 4.2.2)to determine its effect on quality, and the extent to which it hides the influence of upstream variables.

### *4.1.2 Multicollinearity Analysis*

Multicollinearity is a term that refers to the correlation between input variables in a statistical model. In other words, multicollinearity occurs when inputs are not orthogonal. The primary concern with having dependencies among model inputs is overfitting. Overfitting is a condition where there are more variables than orthogonal inputs that the model attempts to fit. The result of overfitting is that the model predicts poorly for unknown data while fits

Table 4.4: Mean and Variance of Integration/Test Phase Variables

| Model Variable | Samples | Mean | Variance |
|---|---|---|---|
| Test Level 1 | 28 | 0.2857 | 0.1446 |
| Test Level 2 | 28 | 0.3393 | 0.1656 |
| Test Level 3 | 28 | 0.1607 | 0.1021 |
| Test Level 4 | 28 | 0.2143 | 0.1684 |
| Develop Integration Strategy | 28 | 0.2143 | 0.1684 |
| Develop Regression Strategy | 28 | 0.2143 | 0.1684 |
| Integrate Software Units | 28 | 1.0000 | 0.0000 |
| Develop Integration Tests | 28 | 0.0000 | 0.0000 |
| Test Integrated Units | 28 | 0.0000 | 0.0000 |
| Regression Test Integrated Units | 28 | 0.2143 | 0.1684 |
| Develop System Test Strategy | 28 | 1.0000 | 0.0000 |
| Develop System Tests | 28 | 0.7857 | 0.1684 |
| Test System | 28 | 1.0000 | 0.0000 |
| Regression Test System | 28 | 0.2143 | 0.1684 |
| Test Completeness | 28 | 0.8018 | 0.0565 |

well to the set of known data inputs. Multicollinearity can be easily detected by identifying correlations between input variables in the model, and eliminating redundant variables.

This section reports the results of correlating the set of inputs used in this software quality model in order to identify those inputs that are not orthogonal. These correlations are calculated using the entire set of software engineering data acquired for this research effort. The intent behind this analysis is to eliminate multicollinearity in the model, and streamline the model structure. All correlations are calculated using Equation 3.10 described in Section 3.3.2.2.

#### *4.1.2.1 Multicollinearity Within Life Cycle Phases*

For the Intuitive and Refined Model structures, model inputs are grouped by phase and related to variables representing completeness and correctness in that phase. In this section, a multicollinearity analysis is performed for model inputs within each life cycle phase. The aim of this analysis is to identify those causal factors that can be consolidated because the represent the same effect. The result is a more succinct model structure that is comprised of orthogonal inputs.

Table 4.5 itemizes the correlation values between inputs to the Requirements Phase. An inspection of the resulting correlation coefficients reveals an opportunity for consolidation of variables between three of the requirements phase practices: Determine Environmental Impact, Develop Release Strategy, and Communicate Software Requirements. Interestingly, while correlated with each other, each of these processes also are correlated with Requirements Level 4 variable. In other words, the presence of experts in eliciting and developing requirements is associated with the actions of planning for the potential impact of the requirements on the operating environment, prioritizing the requirements and mapping their release, and establishing effective communication mechanisms with project stakeholders. These practices could therefore be characterized as indicators of a more mature requirements development team.

The correlation values for input variables that comprise the design phase are captured in Table 4.6. Not surprisingly, the practices of developing an architecture, a detailed design,

Table 4.5: Correlation of Requirements Phase Inputs

| Variable | Req | Req | Req | Req | Eval | Updt | Spec | Det | Dev | Dev | Eval | Com |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Lvl | Lvl | Lvl | Lvl | With | Reqs | SW | Env | Val | Rlse | SW | SW |
| | 1 | 2 | 3 | 4 | Cust | | Reqs | Imp | Crit | Stgy | Reqs | Reqs |
| ReqLvl1 | 1.00 | | | | | | | | | | | |
| ReqLvl2 | -0.42 | 1.00 | | | | | | | | | | |
| ReqLvl3 | -0.37 | -0.65 | 1.00 | | | | | | | | | |
| ReqLvl4 | -0.28 | -0.36 | 0.38 | 1.00 | | | | | | | | |
| EvalCust | -0.18 | 0.11 | -0.09 | 0.45 | 1.00 | | | | | | | |
| UpdtReqs | -0.25 | -0.32 | 0.34 | 0.89 | 0.40 | 1.00 | | | | | | |
| SpecReqs | 0.15 | 0.19 | -0.20 | -0.53 | -0.24 | -0.59 | 1.00 | | | | | |
| DetEnvIm | -0.28 | -0.36 | 0.38 | 1.00 | 0.45 | 0.89 | -0.53 | 1.00 | | | | |
| DevValCr | 0.28 | 0.36 | -0.38 | -1.00 | -0.45 | -0.89 | 0.53 | -1.00 | 1.00 | | | |
| DevRelSt | -0.28 | -0.36 | 0.38 | 1.00 | 0.45 | 0.89 | -0.53 | 1.00 | -1.00 | 1.00 | | |
| EvalReqs | 0.13 | 0.36 | -0.42 | -0.39 | 0.49 | -0.35 | 0.21 | -0.39 | 0.39 | -0.39 | 1.00 | |
| ComReq | -0.28 | -0.36 | 0.38 | 1.00 | 0.45 | 0.89 | -0.53 | 1.00 | -1.00 | 1.00 | -0.39 | 1.00 |

and the design of interfaces are all correlated. Intuitively, this is reasonable. A project that requires some level of documentation of the design typically requires a complete design. As the design practice variables model the same thing, they will be consolidated and represented in the model as a single variable, as described in Section 4.1.2.3. An analysis of the input variable correlations in the Implementation Phase (see Table 4.7) does not reveal any multicollinearity.

Table 4.8 captures the analysis of multicollinearity for model variables in the Integration and Test Phase. Similar to the correlations that were revealed for the requirements phase, there is a set of Integration and Test practices that is associated with the presence of Integration and Test experts (Test Level 4). In this case, the practices that deal with developing a strategy for integration, and also the practices surrounding the planning and execution of regression testing. It would seem that less skill/experience in Integration and Test would be less likely to consider these practices as part of the testing regiment. Because of their

Table 4.6: Correlation of Design Phase Inputs

| Variable | Dsgn | Dsgn | Dsgn | Dsgn | Dev | Dev | Dsgn | Vrfy | Dsgn | DIT | Dsgn | Req |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Lvl | Lvl | Lvl | Lvl | SW | Detl | SW | Dsgn | Trac | | Expn | Vol |
| | 1 | 2 | 3 | 4 | Arch | Dsgn | I/Fs | | | | | |
| DsgnLvl1 | 1.00 | | | | | | | | | | | |
| DsgnLvl2 | -0.41 | 1.00 | | | | | | | | | | |
| DsgnLvl3 | -0.32 | -0.70 | 1.00 | | | | | | | | | |
| DsgnLvl4 | -0.19 | -0.41 | 0.31 | 1.00 | | | | | | | | |
| DevArch | 0.22 | 0.31 | -0.33 | -0.66 | 1.00 | | | | | | | |
| DevDetDs | 0.22 | 0.31 | -0.33 | -0.66 | 1.00 | 1.00 | | | | | | |
| DsgnIFs | 0.22 | 0.31 | -0.33 | -0.66 | 1.00 | 1.00 | 1.00 | | | | | |
| VrfyDsgn | 0.10 | -0.16 | -0.06 | 0.50 | -0.35 | -0.35 | -0.35 | 1.00 | | | | |
| DsgnTrac | 0.26 | 0.29 | -0.50 | -0.21 | 0.30 | 0.30 | 0.30 | 0.49 | 1.00 | | | |
| DIT | -0.39 | -0.02 | 0.21 | 0.48 | -0.51 | -0.51 | -0.51 | 0.21 | -0.07 | 1.00 | | |
| DsgnExpn | -0.04 | 0.29 | -0.17 | -0.40 | 0.32 | 0.32 | 0.32 | -0.41 | -0.06 | -0.38 | 1.00 | |
| ReqVol | 0.27 | 0.07 | -0.23 | -0.27 | 0.22 | 0.22 | 0.22 | -0.10 | 0.07 | -0.42 | -0.02 | 1.00 |

Table 4.7: Correlation of Implementation Phase Inputs

| Variable | Imp | Imp | Imp | Imp | Vrfy | Dev | SW | Dsgn |
|---|---|---|---|---|---|---|---|---|
| | Lvl | Lvl | Lvl | Lvl | SW | Unit | Unit | Vol |
| | 1 | 2 | 3 | 4 | Unit | Tsts | Trac | |
| ImpLvl1 | 1.00 | | | | | | | |
| ImpLvl2 | -0.34 | 1.00 | | | | | | |
| ImpLvl3 | -0.32 | -0.50 | 1.00 | | | | | |
| ImpLvl4 | -0.19 | -0.51 | -0.08 | 1.00 | | | | |
| VrfyUnit | 0.18 | 0.06 | -0.35 | 0.12 | 1.00 | | | |
| DevUnTst | -0.20 | -0.50 | -0.05 | 0.96 | 0.04 | 1.00 | | |
| UnitTrac | 0.27 | 0.28 | -0.34 | -0.29 | 0.80 | -0.39 | 1.00 | |
| DsgnVol | 0.30 | 0.06 | -0.22 | -0.15 | 0.11 | -0.30 | 0.25 | 1.00 |

multicollinearity, the variables representing Test experts and the Integration and Regression Testing practices will be consolidated into a single model variable as described in Section 4.1.2.3.

Consider the model structure that represents the Intuitive Model and the Refined Model (Figure 3.4 in Section 3.2.5.2). In order to completely analyze the effects of multicollinearity in the model, the correlations between model variables that comprise the Intermediary Tier of

Table 4.8: Correlation of Integration/Test Phase Inputs

| Variable | Tst | Tst | Tst | Tst | Dev | Dev | Reg | Dev | Reg |
|---|---|---|---|---|---|---|---|---|---|
| | Lvl | Lvl | Lvl | Lvl | Int | Reg | Tst | Sys | Tst |
| | 1 | 2 | 3 | 4 | Stgy | Stgy | Int | Tst | Sys |
| TstLvl1 | 1.00 | | | | | | | | |
| TstLvl2 | -0.35 | 1.00 | | | | | | | |
| TstLvl3 | -0.24 | -0.30 | 1.00 | | | | | | |
| TstLvl4 | -0.39 | -0.44 | -0.26 | 1.00 | | | | | |
| IntStgy | -0.39 | -0.44 | -0.26 | 1.00 | 1.00 | | | | |
| RegStgy | -0.39 | -0.44 | -0.26 | 1.00 | 1.00 | 1.00 | | | |
| RegTstInt | -0.39 | -0.44 | -0.26 | 1.00 | 1.00 | 1.00 | 1.00 | | |
| DevSysTst | 0.39 | 0.44 | 0.26 | -1.00 | -1.00 | -1.00 | -1.00 | 1.00 | |
| RegTstSys | -0.39 | -0.44 | -0.26 | 1.00 | 1.00 | 1.00 | 1.00 | -1.00 | 1.00 |

the structure are presented in Table 4.9. As shown, measures of Requirements Completeness are correlated with measures of Test Completeness. In other words, the extent to which each project captured the customer needs in requirements was the extent to which those needs were tested for correct operation. Thus, in the Intuitive and Refined model structures, these variables are combined.

Table 4.9: Correlation of Correctness/Completeness Variables

| Variable | Req | Des | Imp | Req | Des | Int/Test |
|---|---|---|---|---|---|---|
| | Correct | Correct | Correct | Complete | Complete | Complete |
| ReqCorrect | 1.00 | | | | | |
| DesCorrect | 0.17 | 1.00 | | | | |
| ImpCorrect | 0.04 | 0.54 | 1.00 | | | |
| ReqComplete | -0.14 | 0.70 | 0.24 | 1.00 | | |
| DesComplete | -0.08 | -0.15 | 0.13 | -0.25 | 1.00 | |
| TstComplete | -0.14 | 0.70 | 0.24 | 1.00 | -0.25 | 1.00 |

#### *4.1.2.2 Multicollinearity Analysis Across Life Cycle Phases*

In Section 4.1.2.1, a multicollinearity analysis was performed among variables within a life cycle phase in order to eliminate redundant variables in the Intuitive and Refined Model structures. In the case of the Direct Effects Model structure, no phase boundaries exist. Thus, it is appropriate to perform a multicollinearity analysis of input variables across life cycle phases, in order to consolidate those inputs that provide the same information to the model.

Table 4.10 shows the correlation between inputs to the requirements phase and inputs to the design phase. Multicollinearity does exist between variables in these phases. For the process variables, verification in the design phase is correlated to verification activities in the requirements phase, or Evaluating Requirements With Customer. Additionally, traceability activities between the design and requirements phases are correlated. Thus, these variables can be consolidated in the model structure as they represent identical effects.

In Table 4.11, a multicollinearity analysis is performed to identify any variables that may be consolidated between the requirements phase model inputs and the implementation phase model inputs. This table reveals a few opportunities. The variables associated with Expert Requirements Leadership are all correlated to the practice of developing unit tests. In addition, a correlation exists between practices in which artifacts are verified in the two phases. That is, in the software engineering data, all projects that practiced evaluating the requirements with the customer also practiced the verification of the software units. Finally,

Table 4.10: Correlation of Requirements Phase Inputs to Design Phase Inputs

| Variable | Dsgn | Dsgn | Dsgn | Dsgn | Dev | Dev | Dsgn | Vrfy | Dsgn | DIT | Dsgn | Req |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Lvl | Lvl | Lvl | Lvl | SW | Detl | SW | Dsgn | Trac | | Expn | Vol |
| | 1 | 2 | 3 | 4 | Arch | Dsgn | I/Fs | | | | | |
| ReqLvl1 | 0.59 | -0.07 | -0.36 | -0.20 | 0.22 | 0.22 | 0.22 | -0.18 | 0.13 | -0.32 | 0.25 | -0.09 |
| ReqLvl2 | -0.09 | 0.45 | -0.33 | -0.37 | 0.28 | 0.28 | 0.28 | 0.11 | 0.36 | -0.01 | 0.11 | 0.32 |
| ReqLvl3 | -0.37 | -0.33 | 0.59 | 0.37 | -0.30 | -0.30 | -0.30 | -0.09 | -0.42 | 0.16 | -0.24 | -0.21 |
| ReqLvl4 | -0.28 | -0.39 | 0.42 | 0.84 | -0.78 | -0.78 | -0.78 | 0.45 | -0.39 | 0.50 | -0.40 | -0.25 |
| EvalCust | 0.10 | -0.16 | -0.06 | 0.50 | -0.35 | -0.35 | -0.35 | 1.00 | 0.49 | 0.21 | -0.41 | -0.10 |
| UpdtReqs | -0.25 | -0.35 | 0.38 | 0.75 | -0.88 | -0.88 | -0.88 | 0.40 | -0.35 | 0.50 | -0.36 | -0.25 |
| SpecReqs | 0.15 | 0.21 | -0.23 | -0.45 | 0.68 | 0.68 | 0.68 | -0.24 | 0.21 | -0.35 | 0.22 | 0.16 |
| DetEnvIm | -0.28 | -0.39 | 0.42 | 0.84 | -0.78 | -0.78 | -0.78 | 0.45 | -0.39 | 0.50 | -0.40 | -0.25 |
| DevValCr | 0.28 | 0.39 | -0.42 | -0.84 | 0.78 | 0.78 | 0.78 | -0.45 | 0.39 | -0.50 | 0.40 | 0.25 |
| DevRelSt | -0.28 | -0.39 | 0.42 | 0.84 | -0.78 | -0.78 | -0.78 | 0.45 | -0.39 | 0.50 | -0.40 | -0.25 |
| EvalReqs | 0.26 | 0.29 | -0.50 | -0.21 | 0.30 | 0.30 | 0.30 | 0.49 | 1.00 | -0.07 | -0.06 | 0.07 |
| ComReq | -0.28 | -0.39 | 0.42 | 0.84 | -0.78 | -0.78 | -0.78 | 0.45 | -0.39 | 0.50 | -0.40 | -0.25 |

the practices of artifact traceability in both phases was performed in tandem. This, in conjunction with the design traceability correlation identified in Table 4.10 is an opportunity for consolidation of all three traceability variables.

Table 4.11: Correlation of Requirements Phase Inputs to Implementation Phase Inputs

| Variable | Imp | Imp | Imp | Imp | Vrfy | Dev | SW | Dsgn |
|---|---|---|---|---|---|---|---|---|
| | Lvl | Lvl | Lvl | Lvl | SW | Unit | Unit | Vol |
| | 1 | 2 | 3 | 4 | Unit | Tsts | Trac | |
| ReqLvl1 | 0.56 | 0.07 | -0.35 | -0.28 | 0.01 | -0.28 | 0.13 | 0.22 |
| ReqLvl2 | -0.18 | 0.60 | -0.26 | -0.37 | 0.19 | -0.36 | -0.36 | -0.01 |
| ReqLvl3 | -0.25 | -0.61 | 0.66 | 0.40 | -0.25 | 0.38 | -0.42 | -0.10 |
| ReqLvl4 | -0.20 | -0.50 | -0.05 | 0.96 | 0.04 | 1.00 | -0.39 | -0.30 |
| EvalCust | 0.10 | -0.20 | -0.31 | 0.52 | 0.66 | 0.45 | 0.49 | 0.02 |
| UpdtReqs | -0.18 | -0.45 | -0.04 | 0.86 | 0.13 | 0.89 | -0.35 | -0.27 |
| SpecReqs | 0.11 | 0.27 | 0.03 | -0.51 | -0.30 | -0.53 | 0.21 | 0.16 |
| DetEnvIm | -0.20 | -0.50 | -0.05 | 0.96 | 0.04 | 1.00 | -0.39 | -0.30 |
| DevValCr | 0.20 | 0.50 | 0.05 | -0.96 | -0.04 | -1.00 | 0.39 | 0.30 |
| DevRelSt | -0.20 | -0.50 | -0.05 | 0.96 | 0.04 | 1.00 | -0.39 | -0.30 |
| EvalReqs | 0.27 | 0.28 | -0.35 | -0.29 | 0.80 | -0.39 | 1.00 | 0.25 |
| ComReq | -0.20 | -0.50 | -0.05 | 0.96 | 0.04 | 1.00 | -0.39 | -0.30 |

The correlation values between variables in the requirements phase and test phase are shown in Table 4.12. As can be seen, multicollinearity exists between requirements phase and test phase inputs. The set of requirements model inputs associated with the consolidated Expert Requirements Leadership variable and the set of integration/test model inputs associated with the consolidated Expert Integration/Test Leadership variable are correlated. Thus, these variables can be further consolidated. In addition, a relationship exists between the Develop Validation Criteria and Develop System Tests model inputs.

Table 4.12: Correlation of Requirements Phase Inputs to Integration/Test Phase Inputs

| Variable | Tst | Tst | Tst | Tst | Dev | Dev | Reg | Dev | Reg |
|---|---|---|---|---|---|---|---|---|---|
| | Lvl | Lvl | Lvl | Lvl | Int | Reg | Tst | Sys | Tst |
| | 1 | 2 | 3 | 4 | Stgy | Stgy | Int | Tst | Sys |
| ReqLvl1 | 0.66 | -0.19 | -0.19 | -0.28 | -0.28 | -0.28 | -0.28 | 0.28 | -0.28 |
| ReqLvl2 | -0.27 | 0.66 | -0.05 | -0.36 | -0.36 | -0.36 | -0.36 | 0.36 | -0.36 |
| ReqLvl3 | -0.19 | -0.46 | 0.32 | 0.38 | 0.38 | 0.38 | 0.38 | -0.38 | 0.38 |
| ReqLvl4 | -0.39 | -0.44 | -0.26 | 1.00 | 1.00 | 1.00 | 1.00 | -1.00 | 1.00 |
| EvalCust | -0.20 | -0.11 | -0.20 | 0.45 | 0.45 | 0.45 | 0.45 | -0.45 | 0.45 |
| UpdtReqs | -0.35 | -0.39 | -0.23 | 0.89 | 0.89 | 0.89 | 0.89 | -0.89 | 0.89 |
| SpecReqs | 0.21 | 0.23 | 0.14 | -0.53 | -0.53 | -0.53 | -0.53 | 0.53 | -0.53 |
| DetEnvIm | -0.39 | -0.44 | -0.26 | 1.00 | 1.00 | 1.00 | 1.00 | -1.00 | 1.00 |
| DevValCr | 0.39 | 0.44 | 0.26 | -1.00 | -1.00 | -1.00 | -1.00 | 1.00 | -1.00 |
| DevRelSt | -0.39 | -0.44 | -0.26 | 1.00 | 1.00 | 1.00 | 1.00 | -1.00 | 1.00 |
| EvalReqs | 0.06 | 0.32 | 0.01 | -0.39 | -0.39 | -0.39 | -0.39 | 0.39 | -0.39 |
| ComReq | -0.39 | -0.44 | -0.26 | 1.00 | 1.00 | 1.00 | 1.00 | -1.00 | 1.00 |

Table 4.13 shows the multicollinearity analysis for input variables in the design and implementation phases. Consistent with the results encountered in the analysis of the requirements inputs, a correlation exists between traceability variables in these two phases. This represents the only opportunity for consolidation of variables in comparing these two phases. Similarly, Table 4.14 compares design variables to model inputs from the integration/test phase. None of the correlations in that table warrant variable consolidation.

The multicollinearity analysis of variables in both the implementation and integration and test phases are captured in Table 4.15. Consistent with findings in the other tables, the

Table 4.13: Correlation of Design Phase Inputs to Implementation Phase Inputs

| Variable | Imp<br>Lvl<br>1 | Imp<br>Lvl<br>2 | Imp<br>Lvl<br>3 | Imp<br>Lvl<br>4 | Vrfy<br>SW<br>Unit | Dev<br>Unit<br>Tsts | SW<br>Unit<br>Trac | Dsgn<br>Vol |
|---|---|---|---|---|---|---|---|---|
| DsgnLvl1 | 0.62 | -0.01 | -0.31 | -0.28 | 0.13 | -0.28 | 0.26 | 0.38 |
| DsgnLvl2 | -0.21 | 0.66 | -0.31 | -0.37 | 0.12 | -0.39 | 0.29 | -0.01 |
| DsgnLvl3 | -0.26 | -0.61 | 0.68 | 0.37 | -0.31 | 0.42 | -0.50 | -0.33 |
| DsgnLvl4 | -0.11 | -0.47 | -0.13 | 0.91 | 0.15 | 0.84 | -0.21 | 0.02 |
| DevArch | 0.16 | 0.39 | 0.04 | -0.75 | -0.23 | -0.78 | 0.30 | 0.24 |
| DevDetDs | 0.16 | 0.39 | 0.04 | -0.75 | -0.23 | -0.78 | 0.30 | 0.24 |
| DsgnIFs | 0.16 | 0.39 | 0.04 | -0.75 | -0.23 | -0.78 | 0.30 | 0.24 |
| VrfyDsgn | 0.10 | -0.20 | -0.31 | 0.52 | 0.66 | 0.45 | 0.49 | 0.02 |
| DsgnTrac | 0.27 | 0.28 | -0.35 | -0.29 | 0.80 | -0.39 | 1.00 | 0.25 |
| DIT | -0.07 | -0.33 | -0.02 | 0.56 | 0.20 | 0.50 | -0.07 | -0.22 |
| DsgnExpn | -0.09 | 0.41 | -0.05 | -0.43 | -0.23 | -0.40 | -0.06 | -0.16 |
| ReqVol | -0.01 | 0.33 | -0.15 | -0.29 | -0.04 | -0.25 | 0.07 | 0.17 |

Table 4.14: Correlation of Design Phase Inputs to Integration/Test Phase Inputs

| Variable | Tst<br>Lvl<br>1 | Tst<br>Lvl<br>2 | Tst<br>Lvl<br>3 | Tst<br>Lvl<br>4 | Dev<br>Int<br>Stgy | Dev<br>Reg<br>Stgy | Reg<br>Tst<br>Int | Dev<br>Sys<br>Tst | Reg<br>Tst<br>Sys |
|---|---|---|---|---|---|---|---|---|---|
| DsgnLvl1 | 0.46 | -0.08 | -0.09 | -0.28 | -0.28 | -0.28 | -0.28 | 0.28 | -0.28 |
| DsgnLvl2 | 0.16 | 0.40 | -0.19 | -0.39 | -0.39 | -0.39 | -0.39 | 0.39 | -0.39 |
| DsgnLvl3 | -0.47 | -0.27 | 0.36 | 0.42 | 0.42 | 0.42 | 0.42 | -0.42 | 0.42 |
| DsgnLvl4 | -0.34 | -0.39 | -0.18 | 0.84 | 0.84 | 0.84 | 0.84 | -0.84 | 0.84 |
| DevArch | 0.31 | 0.34 | 0.21 | -0.78 | -0.78 | -0.78 | -0.78 | 0.78 | -0.78 |
| DevDetDs | 0.31 | 0.34 | 0.21 | -0.78 | -0.78 | -0.78 | -0.78 | 0.78 | -0.78 |
| DsgnIFs | 0.31 | 0.34 | 0.21 | -0.78 | -0.78 | -0.78 | -0.78 | 0.78 | -0.78 |
| VrfyDsgn | -0.20 | -0.11 | -0.20 | 0.45 | 0.45 | 0.45 | 0.45 | -0.45 | 0.45 |
| DsgnTrac | 0.06 | 0.32 | 0.01 | -0.39 | -0.39 | -0.39 | -0.39 | 0.39 | -0.39 |
| DIT | -0.34 | -0.05 | -0.17 | 0.50 | 0.50 | 0.50 | 0.50 | -0.50 | 0.50 |
| DsgnExpn | 0.20 | 0.18 | 0.04 | -0.40 | -0.40 | -0.40 | -0.40 | 0.40 | -0.40 |
| ReqVol | -0.03 | 0.42 | -0.18 | -0.25 | -0.25 | -0.25 | -0.25 | 0.25 | -0.25 |

Develop Unit Tests variable is completely correlated to the set of variables that comprise the Expert Integration/Test Leadership consolidated variable. This is the only significant correlation between inputs to these phases.

Table 4.15: Correlation of Implementation Phase Inputs to Integration/Test Phase Inputs

| Variable | Tst | Tst | Tst | Tst | Dev | Dev | Reg | Dev | Reg |
|---|---|---|---|---|---|---|---|---|---|
| | Lvl | Lvl | Lvl | Lvl | Int | Reg | Tst | Sys | Tst |
| | 1 | 2 | 3 | 4 | Stgy | Stgy | Int | Tst | Sys |
| ImpLvl1 | 0.65 | -0.29 | -0.15 | -0.20 | -0.20 | -0.20 | -0.20 | 0.20 | -0.20 |
| ImpLvl2 | -0.05 | 0.54 | 0.02 | -0.50 | -0.50 | -0.50 | -0.50 | 0.50 | -0.50 |
| ImpLvl3 | -0.15 | -0.08 | 0.35 | -0.05 | -0.05 | -0.05 | -0.05 | 0.05 | -0.05 |
| ImpLvl4 | -0.42 | -0.37 | -0.26 | 0.96 | 0.96 | 0.96 | 0.96 | -0.96 | 0.96 |
| VrfyUnit | -0.10 | 0.13 | -0.09 | 0.04 | 0.04 | 0.04 | 0.04 | -0.04 | 0.04 |
| DevUnTst | -0.39 | -0.44 | -0.26 | 1.00 | 1.00 | 1.00 | 1.00 | -1.00 | 1.00 |
| UnitTrac | 0.06 | 0.32 | 0.01 | -0.39 | -0.39 | -0.39 | -0.39 | 0.39 | -0.39 |
| DsgnVol | 0.14 | 0.25 | -0.10 | -0.30 | -0.30 | -0.30 | -0.30 | 0.30 | -0.30 |

#### *4.1.2.3 Summary of Model Variable Consolidation*

The analysis of the initially proposed model structure has revealed opportunities for consolidation of model variables due to multicollinearity. In all cases, the variables that were consolidated were found to be highly correlated in the Multicollinearity analysis performed in Section 4.1.2. That is, for all projects used as data, one or more of the variables were essentially modeling the same information. In the Requirements phase, for example, three of the process variables (Communicate Software Requirements, Determine Environmental Impact, and Develop Release Strategy) and one of the skill variables (Requirements Level 4) were correlation with a correlation coefficient of 1.0, the maximum value of a correlation coefficient. Thus, those variables model the same input, and can be consolidated into a single variable, which reduces the complexity of the model.

Table 4.16 lists, for each life cycle phase, the variables that were found to be correlated based on the analyses in Section 4.1.2.1, and the name of the variable that will represent their consolidation. In the requirements phase, requirements experts (Requirements Level

4) brought certain practices that were not present in projects without those experts. The consolidation of these variables is characterized by the variable name "Expert Requirements Leadership" as those practices are indicative of that expert presence. A similar situation exists in the Integration and Test Phase. The presence of integration/test experts were associated with planning and structure in the integration/test process. The consolidated variable in that phase is labeled "Expert Integration/Test Leadership". In the Design phase, all three of the design practices were correlated, and so provide an opportunity for combining variables. The variable "Develop Design" will be used to characterize the combination of architectural, detailed, and interface designs.

Table 4.16: Listing of Consolidated Model Variables by Life Cycle Phase

| Life Cycle Phase | Correlated Model Variables | Consolidated Variable Name |
|---|---|---|
| Requirements | Communicate Software Requirements<br>Determine Environmental Impact<br>Develop Release Strategy<br>Requirements Level 4 | Expert Requirements Leadership |
| Design | Develop Software Architecture<br>Develop Detailed Design<br>Design Interfaces | Develop Design |
| Integration/ Test | Develop Integration Strategy<br>Develop Regression Strategy<br>Regression Test Integrated Units<br>Regression Test System<br>Test Level 4 | Expert Integration/Test Leadership |
| Phase Outputs | Requirements Completeness<br>Test Completeness | Requirements and Test Completeness |

Table 4.17 lists the variables across life cycle phases that were found to be correlated based on the analysis in Section 4.1.2.2, and the name of the variable that will represent

their consolidation. It is encouraging to see themes emerge in the practices across life cycle phases. For example, traceability of work products was a practice that either occurred or did not occur in each project, and was not limited to a specific phase. Thus, because they were highly correlated, the phase-specific practices of traceability were consolidated into a single traceability variable. A similar situation existed in terms of work product verification. Projects that practiced verifying requirements with the customer also practiced verification of design and verification of the implementation through the execution of unit tests. Addressing the consolidated Requirements and Testing Leadership variables identified in Section 4.1.2.1, further analysis revealed that they were correlated, and can be represented as a single variable in the model.

Table 4.17: Listing of Consolidated Model Variables Across Life Cycle Phases

| Life Cycle Phase | Correlated Model Variables | Consolidated Variable Name |
|---|---|---|
| Requirements<br>Design<br>Implementation | Evaluate Requirements<br>Design Traceability<br>Software Unit Traceability | Life Cycle Traceability |
| Requirements<br>Design | Evaluate Requirements w/Customer<br>Verify Design | Requirements/Design Verification |
| Requirements<br>Integration/Test | Develop Validation Criteria<br>Develop System Tests | System Test Strategy |
| Requirements<br>Implementation<br>Integration/Test | Expert Requirements Leadership<br>Develop Unit Tests<br>Expert Integration/Test Leadership | Expert Requirements and Testing Leadership |

The consolidation of these variables applies to all of the model structures proposed in this research. In addition to the three Bayesian models being compared in this research, the variables used in the competing models described in Section 3.3.3 will also be subject to the refinements described in this section. Multicollinearity can negatively affect the predictive accuracy of the model by overfitting to the data because multiple instances of the same measure weights that measure more in the model's fit. In the case of the Least Squares Regression model, it can also render the prediction incalculable because multicollinearity can lead to a nonsingular matrix which cannot perform the inversion operation (produces a determinant of 0) required for prediction.

## 4.2 Analysis of the Intuitive Model Structure

This section provides an analysis of the effects of the various causal factors of skill and experience, development process, and problem complexity, on the quality of the final software product. As part of the development of the Methodology (see Section 3.2.5.2), a software quality model was proposed in Figure 3.4 that captured the intuitive relationship between the measures of software quality and their associated driving factors. The focus of this section is to provide a quantitative analysis of the Intuitive Model structure in order to validate those logical cause-effect relationships and, where possible, eliminate any any irrelevant relationships in order to simplify the model structure. This is the mechanism by which the Refined model structure will be derived from the Intuitive Model Structure.

The analysis and refinement of the model structure is based on the determination of correlation between variables within the model structure, and the validation of each correlation through the application of the Hypothesis Test detailed in Section 3.3.2.2. In Section 4.1.1, the mean and variance of model variables was scrutinized in order to eliminate those variables from the model that are invariant, and therefore provide no influence. Then, the model variables in each life cycle phase were analyzed for any multicollinearity (see Section 4.1.2). The approach to identifying and verifying correlations is now applied to the Intuitive Model structure and a commentary is provided on the expectations and actual results of the influences on the various aspects of software quality. This section concludes with a presentation of the Refined Model structure based on the results of this analysis.

### *4.2.1 Analysis of the Input Tier Model Structure*

This section provides an analysis of the Intuitive Model structure proposed in Section 3.2.5.2 and attempts to verify the cause-effect relationship between input and intermediary model nodes. In the three-tiered Intuitive Model of software quality, this section validates the first stage of relationships: analyzing the effect of model inputs on life cycle phase correctness and completeness. The input nodes are the proposed driving factors of software quality, derived from the skill/experience, process, and problem complexity categories. The intermediary nodes in the model are those nodes that represent correctness and completeness at each phase of the development life cycle. These nodes serve as the discriminating factors in the determination of software quality. The intent of this section is to verify that the logical cause-effect relationships between drivers of software quality, and the discriminators of software quality represented by correctness and completeness nodes are valid. The mechanism for this analysis is establishing a statistical correlation through calculation of a correlation coefficient ans verification of that correlation through a hypothesis test. This approach is inline with Kan's criteria for establishing a cause-effect relationship described in Section 2.3.

The expectation of the analysis of the model structure is that the various causal factors identified in each phase with have a significant, positive correlation with the measures of phase completeness and/or correctness. Correctness in a life cycle phase is a measure of the percentage of the work product for that phase (e.g., requirements specification, design document)that did not require correction after baseline. Similarly, the Completeness of a life cycle phase is a measure of the percentage of the work product for that phase that was originally baselined (not added after baseline). In the case of personnel skill and experience, the expectation is that the correlation with phase correctness/completeness will increase as the skill level increases. That is, the relationship between requirements skill and requirements correctness/completeness improves as the skill level being considered improves. For processes, a positive correlation is expected as each identified process has been deemed important to software project success in both the CMMI Version 1.1 [Ins02] and the ISO/IEC 15504 [ISO98]. Software complexity measures are expected to have an inverse relationship with phase correctness and completeness. That is, as complexity increases, it is expected that quality will decrease.

#### *4.2.1.1 Requirements Phase Analysis*

Table 4.18 captures the correlation of the model's requirements phase variables to the correctness of the requirements. Requirements correctness is measured as the proportion of software requirements against which no faults were found after the requirements baseline. Surprisingly, correctness in the requirements phase is not significantly affected by either requirements skill or the majority of the requirements processes. The Evaluate Software Requirements process, and the Evaluate Requirements With Customer process provide a significant, albeit negative, effect. In the Evaluate Software Requirements process, software requirements are traced to system requirements and/or customer needs, and in the Evaluate Software Requirements With Customer process, the requirements are validated by being communicated to the customer, and revising if necessary. Intuitively, these practices should improve the correctness of the requirements document, and reduce the number of required changes after document release. These results suggest an opposite effect - that these practices reduce the correctness of the requirements specification.

The effects of requirements skill/experience and processes on requirements completeness is shown in Table 4.19. Requirements Completeness is the percentage of final product requirements that were originally baselined in the requirements phase. The influence of the causal factors in the requirements phase were mostly as expected, particularly with respect to the requirements processes. That is, they were positively correlated to the requirements completeness measure.

Table 4.18: Correlation of Requirements Variables to Requirements Correctness

| Model Input | Correctness | Test | Significant? |
|---|---|---|---|
| Variable | Correlation | Stat (t) | (t > 1.6726) |
| Requirements Level 1 | -0.1289 | 0.9549 | No |
| Requirements Level 2 | 0.0233 | 0.1715 | No |
| Requirements Level 3 | 0.0415 | 0.3049 | No |
| Expert Requirements Leadership | 0.1758 | 1.3120 | No |
| Specify Software Requirements | -0.0933 | 0.6890 | No |
| Evaluate Requirements w/Customer | -0.2574 | 1.9572 | Yes |
| Update Requirements | 0.1569 | 1.1676 | No |
| Develop Validation Criteria | -0.1758 | 1.3120 | No |
| Evaluate Software Requirements | -0.4163 | 3.3648 | Yes |

The significant positive effect of evaluating the requirements with the customer is encouraging, despite the seemingly adverse effect this practice had on requirements correctness. One advantage to measuring quality in the requirements phase with both correctness and completeness is that it provides insight into two different types of changes: additions and faults. Requirements that do not sufficiently cover the customer's needs require additions. Requirements that are poorly written, or are simply incorrect are faults. This analysis reveals that customer involvement in the requirements phase significantly reduces the number of additions required to the requirements specification, but adversely effects the quality in the writing and correctness of the requirements. It could be argued that customers are not as skilled in the formal language required to author requirements specifications, and the influence of that lack of requirements skill is reflected in the quality of the specification's statements.

In terms of associating the requirements skill of the software engineering team to requirements completeness, the presence of requirements "experts", or Expert Requirements

Leadership has a significant positive effect on the completeness of the requirements. Recall from Section 4.1.2.3 that this model variables encompasses Requirements Level 4 from the skill variables, and three of the requirements process variables. It is not surprising that a project that has an expert available for the elicitation and specification of software requirements would be more likely to completely capture the customer needs at the time of requirements baseline.

Table 4.19: Correlation of Requirements Variables to Requirements Completeness

| Model Input | Completeness | Test | Significant? |
|---|---|---|---|
| Variable | Correlation | Stat (t) | (t > 1.6726) |
| Requirements Level 1 | -0.0966 | 0.7130 | No |
| Requirements Level 2 | 0.0267 | 0.1961 | No |
| Requirements Level 3 | -0.0463 | 0.3408 | No |
| Expert Requirements Leadership | 0.3493 | 2.7390 | Yes |
| Specify Software Requirements | -0.1479 | 1.0991 | No |
| Evaluate Requirements w/Customer | 0.6013 | 5.5303 | Yes |
| Update Requirements | 0.3121 | 2.4141 | Yes |
| Develop Validation Criteria | -0.3493 | 2.7390 | Yes |
| Evaluate Software Requirements | 0.0083 | 0.0611 | No |

The correlation of requirements causal factors to requirements phase correctness and completeness deviated from what was expected. Only the presence of requirements experts had a significant effect in terms of the teams's collective requirements skill. In terms of process, the results were mixed. Requirements Completeness was significantly affected by 6 of the 8 requirements processes where requirements correctness was only affected by 2 processes. It could be interpreted that requirements process is more suited to identifying all of the customer needs and their associated requirements than it is conducive to expressing those requirements correctly.

#### *4.2.1.2 Design Phase Analysis*

The correlation of factors of design skill, process, and complexity to correctness in the software's design is shown in Table 4.20. Design correctness measures the proportion of design modules in which design faults were not found after the design was baselined. The majority of skill and process factors had a significant influence on design correctness. The variables associated with design size and complexity interestingly had no effect on the faults associated with the design.

The effects of the design skill variables on design correctness are as expected. The increasing presence of more skilled and experienced software designers (Skill Levels 3 and 4) is positively correlated with correctness in the design. Similarly, the increasing presence of less skilled software designers (Skill Level 2) is negatively correlated with design correctness, indicating that more design faults are introduced by these personnel. This premise must be explored further, however, as the same dramatic deviation was not noticed for the correlation values associated with Skill Level 1. In a software development effort, the minimally skilled (e.g., Skill Level 1) are easily recognized, and often assigned more trivial aspects of the design and implementation in order to improve their individual skill/experience. However, those with a marginal skill set or with a small amount of experience (e.g., Skill Level 2) are not as appropriately allocated to easier tasks. A developer with basic competence in design is often assigned an equal share of the development responsibilities with his/her more highly skilled colleagues. It is proposed that those developers that have a small amount of software design skill (Skill Level 2) are given software development tasks that are inappropriately matched to their skill/experience level. For that reason, the increased presence of Skill Level 2 has an adverse negative impact on the quality of the software development product.

While all of the design process variables had a significant impact on the correctness of the design, only the practice of design verification is positively correlated with correctness in the design as expected. The Develop Design process, which is a consolidation of three software design processes (Develop Software Architecture, Develop Detailed Design, and Design Interfaces), focuses on the traditional analysis of requirements and methodical decomposition of the design into manageable and documented components. This practice negatively affected the percentage of designed components that remained unchanged after baseline. This result is unexpected.

Curiously, the measures of design complexity had no significant effect on the correctness of the design. This is a deviation from the expected inverse relationship between complexity and correctness. That is, as design complexity increased, it was expected that the correctness of the design would decrease. This is not too alarming given the relatively simple nature of the underlying software engineering data. These results are based on small-scale software projects with typically less than 30 classes to develop, and few interfaces to manage. While the selected design complexity measures were normalized by design size to provide a basis of comparison, there is no highly complex or large-scale software engineering measures available in this data set. Thus, it is not surprising that the effect of complexity is limited by the lack of complexity in the underlying data set.

Table 4.20: Correlation of Design Variables to Design Correctness

| Model Input<br>Variable | Correctness<br>Correlation | Test<br>Stat (t) | Significant?<br>(t > 1.7010) |
|---|---|---|---|
| Design Level 1 | 0.0454 | 0.2317 | No |
| Design Level 2 | -0.5128 | 3.0461 | Yes |
| Design Level 3 | 0.4325 | 2.4456 | Yes |
| Design Level 4 | 0.4134 | 2.3153 | Yes |
| Develop Design | -0.3758 | 2.0677 | Yes |
| Verify Design | 0.5092 | 3.0164 | Yes |
| Design Traceability | -0.4184 | 2.3489 | Yes |
| Depth of Inheritance Tree | -0.2589 | 1.3665 | No |
| Design Expansion | -0.2040 | 1.0627 | No |
| Interface Format Expansion | 0.0868 | 0.3896 | No |
| Interface Protocol Expansion | -0.3367 | 1.5989 | No |
| Requirements Volatility | -0.0713 | 0.3646 | No |
| Requirements Correctness | 0.1078 | 0.5531 | No |

Design completeness is the the proportion of the originally baselined design that is present in the software's final design. Table 4.21 captures the correlation of the design phase input variables to design completeness. The skill of the software designers had little effect on the design completeness, save for the negative impact of those personnel with Design Level 2. As with Design Correctness, this influence is attributed to the inappropriate assignment of design tasks to these developers with marginal design skill and experience.

From a design process perspective, only two of the practices had a measurable effect on Design Completeness. As with Design Correctness, the Verify Design process was positively correlated with the completeness of the design. This is intuitively expected as explicitly verifying the design should allow the development team to more easily identify requirements and customer expectations that were unmet. The Design Traceability practice, in which elements of the design are traced to software requirements has an unexpected negative correlation with Design Completeness. Logically, this practice should increase a development team's ability to account for all requirement, and the reason for these results is unknown.

Table 4.21: Correlation of Design Variables to Design Completeness

| Model Input<br>Variable | Completeness<br>Correlation | Test<br>Stat (t) | Significant?<br>(t > 1.6970) |
|---|---|---|---|
| Design Level 1 | 0.0454 | 0.5178 | No |
| Design Level 2 | -0.5128 | 2.0481 | Yes |
| Design Level 3 | 0.4325 | 1.3896 | No |
| Design Level 4 | 0.4134 | 1.4686 | No |
| Develop Design | -0.3758 | 0.9372 | No |
| Verify Design | 0.5092 | 2.0672 | Yes |
| Design Traceability | -0.4184 | 2.5310 | Yes |
| Depth of Inheritance Tree | -0.2589 | 1.0393 | No |
| Design Expansion | -0.3653 | 2.0764 | Yes |
| Interface Format Expansion | -0.5571 | 3.1463 | Yes |
| Requirements Volatility | -0.0713 | 0.8089 | No |
| Requirements Completeness | 0.7276 | 5.6121 | Yes |

An inverse relationship exists between design complexity variables that were found to be a significant influence, and design completeness. This is intuitive as an increase in complexity should logically result in a decrease in quality. It should be noted that the effect of volatility in the requirements was virtually negligible in terms of both design completeness and correctness. Upstream changes to the requirements document would seem to have an effect of the quality of the design. This could be the result of relatively minor changes in the requirements after baseline.

In terms of significance, Design Correctness is affected most by the skill of the design team, and by the practice of design verification. Design Completeness is affected most strongly by completeness in the requirements phase, and also is significantly affected by the Verify Design practice. The value of reviewing a developed software engineering artifact has surfaced in the analysis of both the requirements and design phase correctness and completeness variables. The review practices for both phases have the most significant impact on the quality of these artifacts from a process perspective.

#### *4.2.1.3 Implementation Phase Analysis*

Implementation is the phase of software development where the design is transformed into an operational software product. Correctness in this phase refers to the proportion of baselined source code that was unchanged by any discovered faults. Table 4.22 captures the effects of the identified causal factors of the Implementation Phase on the Implementation Correctness value. Almost all of the variables that characterize skill, process, and complexity in the implementation phase were found to be a significant influence on the Implementation Correctness.

The effects of implementation skill on correctness are as expected with the more skilled developers having a positive correlation with correctness, and the less skilled developers having a negative correlation with correctness. The interpretation of these relationships is that an increasing presence of more skilled developers increases the correctness of the source code and decreases the number of code faults while an increasing presence of less skilled developers decreases the correctness and increases the number of code faults. Of particular note is the unusually high negative correlation of those personnel with Skill Level

2 to the correctness of the implementation. Consistent with the results found in the analysis of design factors in Section 4.2.1.2, this is likely the result of software developers with a marginal implementation skill set being assigned implementation tasks that are inconsistent with their skill level.

Table 4.22: Correlation of Implementation Variables to Correctness

| Model Input<br>Variable | Correctness<br>Correlation | Test<br>Stat (t) | Significant?<br>(t > 1.7010) |
|---|---|---|---|
| Implementation Level 1 | -0.3768 | 2.0743 | Yes |
| Implementation Level 2 | -0.7576 | 5.9192 | Yes |
| Implementation Level 3 | 0.3673 | 2.0138 | Yes |
| Implementation Level 4 | 0.5968 | 3.7925 | Yes |
| Develop Unit Tests | 0.6452 | 4.3064 | Yes |
| Verify Software Units | -0.2098 | 1.0940 | No |
| Software Unit Traceability | -0.6421 | 4.2707 | Yes |
| Design Volatility | -0.4400 | 2.4982 | Yes |
| Design Correctness | 0.6639 | 4.5263 | Yes |

The effects of Implementation Phase process on implementation correctness is surprising. In the Requirements and Design life cycle phases, practices that involved the verification of work products had a significant and (often) positive effect on the correctness and completeness characterized in each phase. In implementation, the practice of verification is not a significant influence, yet the development of unit tests is highly correlated to correctness in the source code implementation. It is proposed that the Verify Software Units practice was too vague in its description as verification can mean both unit testing and peer review. The development of unit tests is positively correlated to implementation correctness, and so it is reasonable to infer that execution of those tests also positively affects correctness (as it is unlikely that unit test were developed and not executed. Unfortunately, the significance of the "Verify Software Units" practice it seems was adversely affected by an ambiguous definition.

#### *4.2.1.4 Integration/Test Phase Analysis*

Table 4.23 captures the correlation between the skill and process variables in the Integration/Test phase of the software life cycle, and the Test Completeness measure. Test Completeness measures the percentage of requirements that are functionally verified in the software test. From a skill perspective, only Expert Integration/Test Leadership (a combination of skill and process variables), has a significant positive effect on the completeness of the test. Thus the increasing presence of integration and testing experts, and the processes associated with those experts, increases the quality of the testing artifacts in the form of the requirements coverage of the software tests. Similar to the results from the analysis of the Requirements Phase in Section 4.2.1.1, the presence of experts is an indication of leadership within the testing team, and is reasonable to infer as the driving factor.

Table 4.23: Correlation of Integration/Test Variables to Completeness

| Model Input<br>Variable | Completeness<br>Correlation | Test<br>Stat (t) | Significant?<br>($t > 1.6726$) |
|---|---:|---|---|
| Test Level 1 | -0.0400 | 0.2941 | No |
| Test Level 2 | -0.1505 | 1.1188 | No |
| Test Level 3 | -0.2091 | 1.5717 | No |
| Expert Integration/Test Leadership | 0.3493 | 2.7390 | Yes |
| Develop System Tests | -0.3493 | 2.7390 | Yes |

### *4.2.2 Analysis of the Intermediary Tier Model Structure*

This section analyzes the second stage of the three-tier Bayesian model for software quality shown in Figure 3.4. The intent of this analysis is to validate the proposed cause-effect relationship between correctness and completeness in each life cycle phase, and each modeled software quality variable. Phase correctness is measured as the percentage of phase products that were produced, and remained unchanged due to faults after phase completion. Phase completeness is measured as the proportion of the final phase products that were originally baselined. The intended focus of this analysis is a determination of significance. That is, what is level of influence of the correctness and completeness measures of each life cycle phase on the measured elements of software quality.

Intuitively, it is expected that phase correctness and completeness will be positively correlated with measures of software quality. That is, for increasing values of correctness and completeness in any life cycle phase, it is expected that a given quality measure will also increase in value. This implies that writing correct and complete artifacts for that phase had an influence on the quality metric being considered. Negative correlations have different meanings for each correctness/completeness value, and will be discussed as they occur in the subsections below. It should also be noted that for some software quality variables, the various measures of phase completeness and correctness did not vary in value across the project samples, and thus have no effect in predicting values of software quality. In these cases, the software quality variables were simply not included in the correlation tables as they added no value to the results.

#### *4.2.2.1 Effects of Requirements Phase Correctness and Completeness*

Table 4.24 captures the correlation and significance test results for the effect of requirements correctness on the various measures of software quality. Requirements correctness was found to be a significant influence on 5 of the software quality variables, and positively correlated for each of these. This implies that a development team's ability to accurately capture and baseline requirements has a positive effect on these 5 software quality variables. Functional Adequacy, Data Encryption, and Efficiency Compliance were all highly influenced by correctness in the requirements.

Completeness in the requirements phase reflects the ability of the development team to adequately address the customer's needs in the initial baseline. The correlation values and associated significance between software quality variables and Requirements Completeness is captured in Table 4.25. Of the 25 software quality measures evaluated in this analysis, 11 were found to be significantly influenced by requirements completeness.

The expectation of the effect of requirements completeness intuitively is that it would have a positive correlation to a given software quality variable, thereby indicating that establishing a complete initial requirements baseline. This gives a development team a consistent foundation of analyzed customer needs and expectations that changes little, and thus allows

Table 4.24: Correlation of Requirements Correctness to Software Product Quality

| Software<br>Quality Variable | No.<br>Samps | Correctness<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | 0.9967 | 8.83E1 | 1.6726 | Yes |
| Functional Completeness | 28 | -0.1509 | 1.1218 | 1.6726 | No |
| Functional Coverage | 28 | 0.1278 | 0.9469 | 1.6726 | No |
| Specification Stability | 28 | 0.1863 | 1.3937 | 1.6726 | No |
| Accuracy | 9 | -0.3001 | 1.2585 | 1.7340 | No |
| Precision | 9 | -0.1170 | 0.4714 | 1.7340 | No |
| Data Exchangeability | 13 | 0.2631 | 1.3358 | 1.7060 | No |
| Interface Consistency | 9 | 0.4851 | 2.2188 | 1.7340 | Yes |
| Access Auditability | 7 | 0.6455 | 2.9277 | 1.7610 | Yes |
| Data Encryption | 2 | 1.0000 | N/A | 2.1320 | Yes |
| Functional Compliance | 5 | 0.2811 | 0.8286 | 1.8120 | No |
| **Efficiency** | | | | | |
| Efficiency Compliance | 3 | 1.0000 | 1.34E8 | 1.9430 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | -0.1505 | 0.7457 | 1.7060 | No |
| **Portability** | | | | | |
| Software Operability | 13 | 0.0037 | 0.0181 | 1.7060 | No |
| Hardware Operability | 7 | -0.3037 | 1.1923 | 1.7460 | No |

for consistency in the downstream phases. While 6 of the software quality measures were determined to be positively correlated to requirements completeness, 5 of them were negatively correlated. A negative correlation does not imply that requirements completeness adversely impacted quality. Rather, a negative correlation is an indication of the difficulty in gathering requirements in that particular quality area.

Recall from the Multicollinearity analysis (Section 4.1.2) that Requirements Completeness and Integration/Test Phase Completeness were combined as variables. Thus, the results captured in Table 4.25 apply to that variables as well. Completeness in the Integration/Test phase describes the extent to which the developed software tests cover the requirements that are to be verified. A positive correlation to a software quality attribute is an indication that the ability to verify that quality need was instrumental in that quality attribute being present in the final software product. A negative correlation implies an inverse relationship: the more comprehensive the test, the less requirements within that quality attribute that are verified. A negative correlation is more a commentary on implementation of that quality attribute than the quality of the test. That is, it is reasonable to infer that a more thorough test would be detrimental to a software product that was poorly implemented, or even an indication of a quality attribute that is difficult to implement as specified.

Consider some of the areas of software quality that are negatively correlated with software quality: Precision, User Cancelability, Failure Avoidance. These are typically the types of details that are rarely captured at the outset by an analysis of the customer's needs, but rather begin to surface as the design matures. The various approaches to avoiding failure, for example, become much more clear as a design begins to shape what kinds of failures might occur. As another example, the requirements for precision of data items evolve as the definition of interfaces evolves during the design process. Thus, it is reasonable to infer that the nature of those 5 software quality attributes lend themselves more to additional requirements after the specification has been baselined, and therefore are negatively and significantly correlated to requirements completeness.

Table 4.25: Correlation of Requirements and Test Completeness to Software Product Quality

| Software | No. | Completeness | Test | Critical | Significant? |
|---|---|---|---|---|---|
| Quality Variable | Samps | Correlation | Stat(t) | Value(cv) | (t > cv) |
| **Suitability** | | | | | |
| Functional Adequacy | 28 | -0.1231 | 0.8945 | 1.6726 | No |
| Functional Completeness | 28 | 1.0000 | N/A | 1.6726 | Yes |
| Functional Coverage | 28 | 0.9616 | 5.0408 | 1.6726 | Yes |
| Specification Stability | 28 | 0.0214 | 0.1571 | 1.6726 | No |
| Accuracy | 9 | 0.1684 | 0.6834 | 1.7340 | No |
| Precision | 9 | -0.5370 | 2.5465 | 1.7340 | Yes |
| Data Exchangeability | 13 | 0.0318 | 0.1556 | 1.7060 | No |
| Interface Consistency | 9 | -0.2692 | 1.1180 | 1.7340 | No |
| Access Auditability | 7 | -0.6752 | 3.1706 | 1.7610 | Yes |
| Access Controllability | 8 | 0.5900 | 2.7341 | 1.7460 | Yes |
| Functional Compliance | 5 | -0.8880 | 5.4632 | 1.8120 | Yes |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | -0.6467 | 3.1721 | 1.7460 | Yes |
| Incorrect Op Avoidance | 5 | -0.2489 | 0.7267 | 1.8120 | No |
| Restorability | 7 | 0.8382 | 5.3251 | 1.7610 | Yes |
| **Efficiency** | | | | | |
| Efficiency Compliance | 3 | -0.0192 | 0.0383 | 1.9430 | No |
| **Usability** | | | | | |
| User Cancelability | 3 | -0.7868 | 3.1225 | 1.8600 | Yes |
| Status Monitoring | 4 | 0.2483 | 0.7251 | 1.8120 | No |
| Usability Compliance | 7 | 0.7705 | 4.1868 | 1.7610 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | 0.4030 | 2.1571 | 1.7060 | Yes |
| **Portability** | | | | | |
| Software Operability | 13 | 0.0088 | 0.0430 | 1.7060 | No |
| Hardware Operability | 7 | 0.1857 | 0.7070 | 1.7460 | No |

#### *4.2.2.2 Effects of Design Phase Correctness and Completeness*

Table 4.26 captures the effects of design correctness on the various indicators of software quality. Recall that design correctness represents the proportion of the baselined design against which no faults were identified. Intuitively, fewer design faults should lead to higher quality, and so the expectation is that any significant correlations would be positive. Two of the system-level quality variables, Functional Implementation Completeness and Coverage, are the only ones significantly positively affected by correctness in the design. This is encouraging in that it reinforces the expectation in terms of the positive effect that correctness in the design has on the quality of the delivered system. However, four other quality variables were found to be significantly influenced by design correctness, but the correlations were negative. The cause for this is unclear.

Table 4.26: Correlation of Design Correctness to Software Product Quality

| Software<br>Quality Variable | No.<br>Samps | Correctness<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | 0.1681 | 0.8355 | 1.7060 | No |
| Functional Completeness | 28 | 0.7536 | 5.8457 | 1.7010 | Yes |
| Functional Coverage | 28 | 0.7662 | 6.0803 | 1.7010 | Yes |
| Specification Stability | 28 | -0.0158 | 0.0804 | 1.7010 | No |
| Precision | 9 | -0.8660 | 3.4641 | 1.9430 | Yes |
| Data Exchangeability | 13 | -0.9751 | 1.52E1 | 1.7610 | Yes |
| Interface Consistency | 9 | -0.0504 | 0.1237 | 1.8600 | No |
| Functional Compliance | 5 | -0.9503 | 8.6330 | 1.8120 | Yes |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | -0.4540 | 1.2481 | 1.8600 | No |
| **Efficiency** | | | | | |
| Efficiency Compliance | 3 | -0.3054 | 0.6415 | 1.9430 | No |
| **Usability** | | | | | |
| Usability Compliance | 7 | 0.1557 | 0.5460 | 1.7610 | No |
| **Portability** | | | | | |
| Software Operability | 13 | -0.6724 | 2.2249 | 1.8600 | Yes |

Four of the software quality variables were found to be significantly related to completeness in the design. Recall that Design Completeness represents the extent to which the baselined software design addresses and fulfills the needs identified by the requirements.

Any significant correlations are expected to be positive, and this was the case for all four of the affected variables.

Table 4.27: Correlation of Design Completeness to Software Product Quality

| Software<br>Quality Variable | No.<br>Samps | Completeness<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | -0.0867 | 0.4437 | 1.6970 | No |
| Functional Completeness | 28 | 0.7276 | 5.6121 | 1.6970 | Yes |
| Functional Coverage | 28 | 0.6897 | 5.0408 | 1.6970 | Yes |
| Specification Stability | 28 | 0.1479 | 0.7914 | 1.6970 | No |
| Data Exchangeability | 13 | 0.1667 | 0.5855 | 1.7610 | No |
| **Usability** | | | | | |
| Usability Compliance | 7 | 0.6455 | 2.9277 | 1.7610 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | -0.2000 | 0.6455 | 1.7820 | No |
| **Portability** | | | | | |
| Software Operability | 13 | 0.5590 | 1.9069 | 1.8120 | Yes |

#### *4.2.2.3 Effects of Implementation Phase Correctness*

The effects of correctness in the implementation phase on the various aspects of software quality are captured in Table 4.28. Implementation correctness is measured as the ratio of implemented software modules without a coding fault identified after the completion of the implementation phase. Thus, a positive correlation with implementation correctness indicates that modules implemented correctly the first time have an influence on the particular quality attribute. A negative correlation implies that those modules that were implemented more than once (either fully or partially) were more likely to produce higher quality results.

Negative correlations are indicators of elements of software quality that are more complex from the perspective of actual coding, and thus are more effectively implemented across multiple attempts. The intended focus of this analysis is a determination of significance. That is, what is the significance of the correctness of the implementation in terms of the measured elements of software quality.

Table 4.28: Correlation of Implementation Correctness to Software Product Quality

| Software<br>Quality Variable | No.<br>Samps | Correctness<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | 0.0363 | 0.1779 | 1.7060 | No |
| Functional Completeness | 28 | 0.6083 | 3.9076 | 1.7010 | Yes |
| Functional Coverage | 28 | 0.5886 | 3.7126 | 1.7010 | Yes |
| Specification Stability | 28 | -0.3558 | 1.9414 | 1.7010 | Yes |
| Precision | 9 | -0.8260 | 2.9312 | 1.9430 | Yes |
| Data Exchangeability | 13 | -0.5521 | 2.2935 | 1.7610 | Yes |
| Interface Consistency | 9 | 0.1005 | 0.2474 | 1.8600 | No |
| Functional Compliance | 5 | -0.2706 | 0.7951 | 1.8120 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.3107 | 0.8006 | 1.8600 | No |
| Incorrect Op Avoidance | 5 | 1.0000 | N/A | 2.1320 | Yes |
| Restorability | 7 | 0.2500 | 0.7303 | 1.8120 | No |
| **Efficiency** | | | | | |
| Efficiency Compliance | 3 | -0.9972 | 2.66E1 | 1.9430 | Yes |
| **Usability** | | | | | |
| Usability Compliance | 7 | 0.4737 | 1.8631 | 1.7610 | Yes |
| **Portability** | | | | | |
| Software Operability | 13 | -0.8470 | 3.9030 | 1.8600 | Yes |

Nine different software quality attributes were significantly affected by the development teams ability to correctly implement the source code. In terms of the system-level quality measures, both Functional Completeness and Coverage were positively influenced by implementation correctness, as were the Incorrect Operation Avoidance and Usability Compliance attributes. The remaining variables were found to be negatively correlated, meaning that their implementation was more complex, and could not be achieved in the original baseline.

### *4.2.3 Structure of the Refined Model*

This section describes the adjustments to the Bayesian Belief Network that establishes the cause-effect relationships that comprise the proposed software quality model. Using the results of the analyses in the previous sections, the Refined Model structure is presented below. It streamlines the model structure by eliminating those intuitive cause-effect relationships that could not be validated empirically. The Refined Model structure is presented as a set of directed acyclic graphs representing the various phase subnets that comprise the model. In addition, each software quality variable is evaluated in terms of the ability of the model's structure to account for causal factors that influence that particular quality attribute.

Figure 4.1 shows the requirements subnet that has been revised to reflect the correlation analysis detailed in Section 4.2.1.1. Requirements Correctness was found to be significantly affected by only two practices: Evaluating the Requirements with the Customer, and Evaluating Requirements through traceability. Requirements Completeness in influenced by a more complete set of requirements phase variables including the presence of requirements development expertise, and a more comprehensive set of requirements practices.

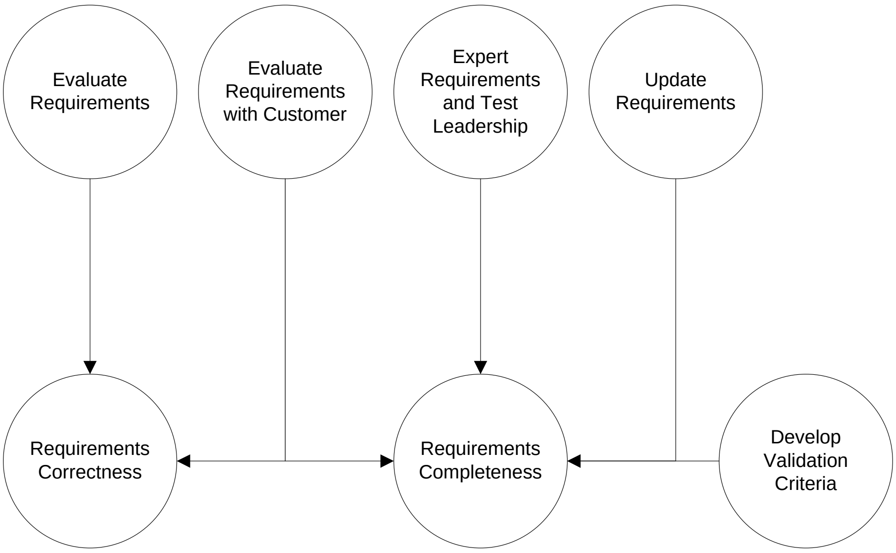
Evaluate Requirements
Evaluate Requirements with Customer
Expert Requirements and Test Leadership
Update Requirements
Requirements Correctness
Requirements Completeness
Develop Validation Criteria

Figure 4.1: Refined Requirements Subnet

The modifications to the structure design subnet, as a result of the analysis of significant variables, are shown in Figure 4.2. Design skill and process variables are well represented as drivers of both design correctness and completeness. One interesting omission is the lack of design complexity metrics as influences on phase quality.

Figure 4.3 captures the subnet for the Implementation phase that has been refined to include only correlated associations. As described above, most of the model inputs for this life cycle phase were found to be significant with respect to the correctness of the implementation. All of the skill variables were found to have an influence, as did the processes that had a variance. Finally, changes to the baselined design were also found to affect the correctness of the implementation.

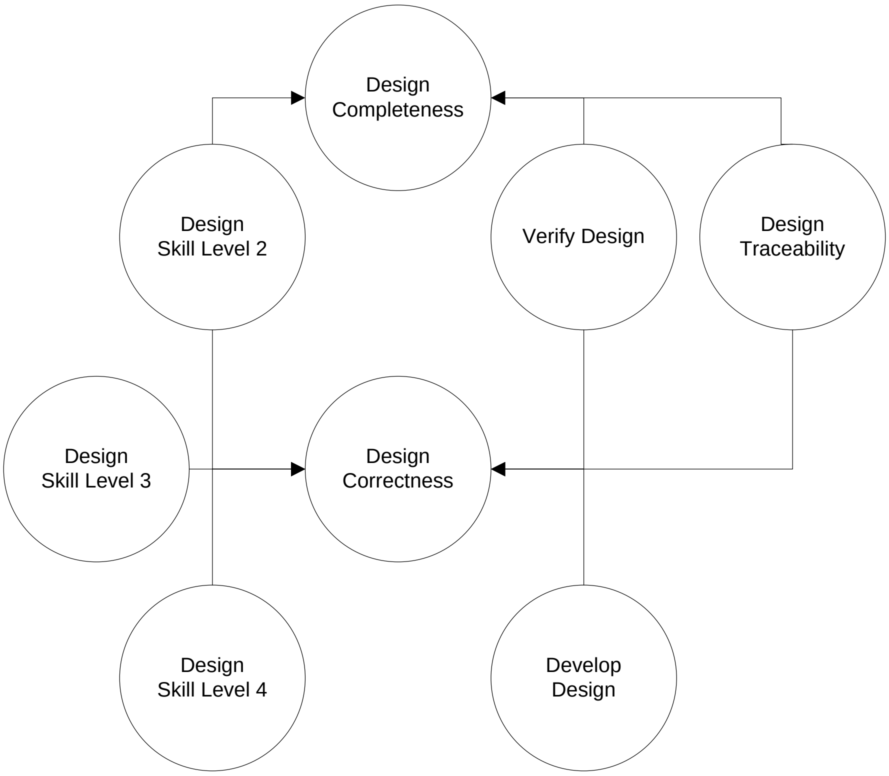

Figure 4.2: Refined Design Subnet

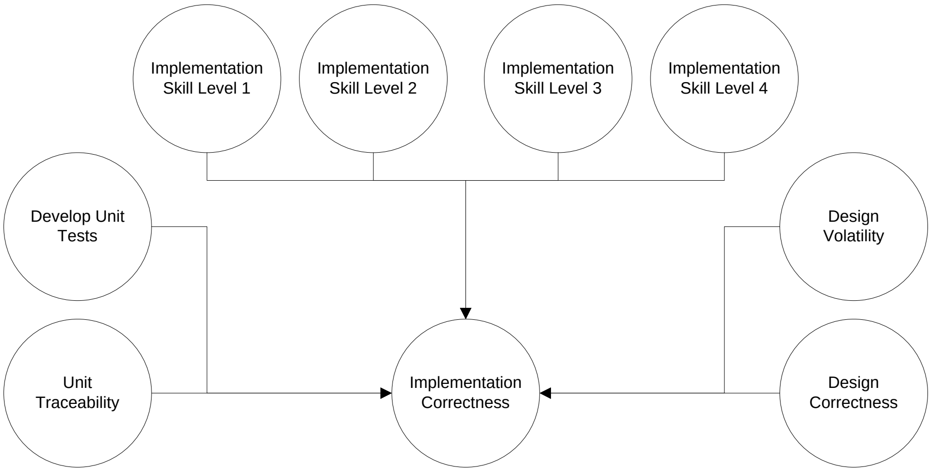
Implementation Skill Level 1
Implementation Skill Level 2
Implementation Skill Level 3
Implementation Skill Level 4
Develop Unit Tests
Design Volatility
Unit Traceability
Implementation Correctness
Design Correctness

Figure 4.3: Refined Implementation Subnet

The Refined model's Integration and Test subnet is shown in Figure 4.4. Many of the practices in this phase were eliminated due to invariance, or combined with the Requirements and Testing expertise consolidated variable. The resultant subnet is comprised of only two model inputs. The Develop System Tests node was the only practice from this life cycle phase that was found to be an independent variable, and have a significant effect on the completeness of the Integration and Test activities.

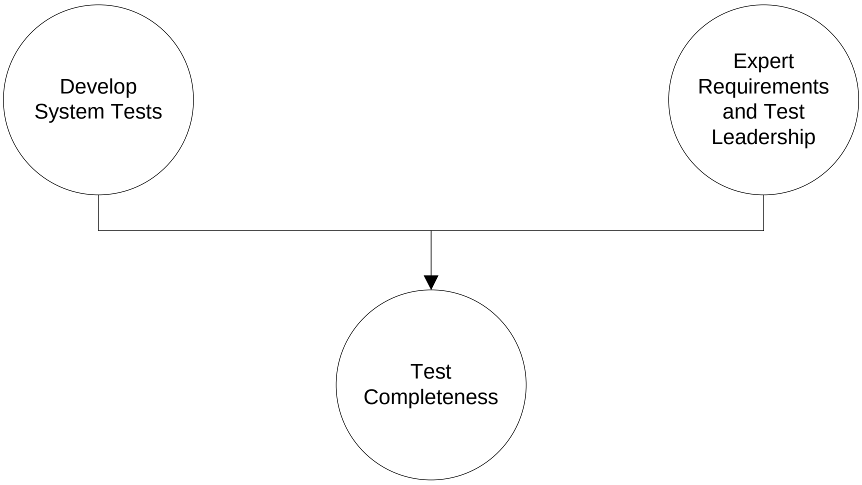

Figure 4.4: Refined Integration/Test Subnet

## 4.3 Analysis of Direct Effects on Software Quality

The analysis of the Intuitive software quality model's structure presented in Section 4.2 revealed that not all of the software quality variables are influenced by the three-tiered structure proposed. For that reason, it is appropriate to analyze the set of software quality variables in an attempt to determine how the various causal factors of software quality (e.g., team skill, process maturity, and problem complexity) directly affect each one. The intent of this analysis is to identify the influences of any causal factors in the model that were masked by the effects of the phase completeness/correctness variables in the intermediary tier. The Intuitive and Refined models structure assume that the software quality variables are most appropriately modeled using the three-tier model structure. This analysis considers the Direct Effects option, in which those software quality variables are modeled by associating the model inputs directly to the outputs in the Bayesian Belief Network. The directed acyclic graph that captures the Direct Effects Model structure is shown in Figure 3.5.

The method for determining the direct effects of the various software quality influences on the software quality variables is the correlation and associated hypothesis test described in Section 4.2. This section is organized by each software quality variable modeled, and provides a graph depicting the significant influences on that particular quality variable, and the degree of each influence. The complete data tables that capture the specifics of the analysis including the correlation values, the calculated Test Statistic, and the Critical Values are found in Appendix B.

By proposing a the Direct Effects Model in addition to the Intuitive and Refined Models, it becomes necessary to determine which approach is more accurate in predicting software quality. Section 4.5, which validates the model's ability to predict for each software quality variable, provides a comparison between the three structure options. In addition, the Model Comparison section provides an accuracy comparison with competing software quality modeling approaches.

### *4.3.1 Analysis of the Functionality Software Quality Attributes*

Functionality is the partition of the ISO/IEC 9126 Software product quality standard that addresses the functional operation of the software product. It covers the suitability of the software system in terms of the captured customer needs and software requirements, the level of quality in the implementation of any defined interfaces, the computational accuracy associated with any developed algorithms, and covers any security needs that are required. The intent of this section is to identify and analyze the elements of development team skill, process maturity, and problem complexity that have a significant influence on the software quality variables associated with Functionality.

#### *4.3.1.1 Suitability*

Suitability is a sub-characteristic of Functionality that addresses the quality of the delivered system as a whole. That is, while most of the other software quality attributes address the extent to which a subset of the customer needs and/or software requirements are met, the Suitability variables address the extent to which the full complement of the customer needs and/or software requirements were met. The sample set for the Suitability variables was the entire project data set.

Three software quality variables were selected to represent the Suitability of the developed software: Functional Adequacy, Functional Implementation Completeness, and Functional Implementation Coverage. Functional Adequacy is a measure of how well the development team implemented the software system, including the extent to which the system was verified in test. Functional Implementation Completeness, as its name implies, is a variable that represents the proportion of software requirements that were verified to have been implemented in the delivered software system. Finally, Functional Implementation Coverage measures the extent to which the software was correctly implemented, and takes into account any faults that were identified yet not corrected before the system was delivered to the customer.

Figure 4.5 depicts the correlation values for the software quality causal factors that had a direct effect on Functional Adequacy. Contrary to expectations, all four factors that were significantly correlated with Functional Adequacy had a negative effect, meaning that the proportion of software requirements verified in test decreased as values of the various influences increased. The negative correlation is reasonable when considering the influence of Test Skill Level 2. It could be inferred that the lack of experience in developing and executing software test had a negative effect on the proportion of requirements tested to be adequate. Three of the factors that were directly correlated to Functional Adequacy were process variables - specifically those processes in which the requirements, design and implementation artifacts were verified and traced to upstream project artifacts.

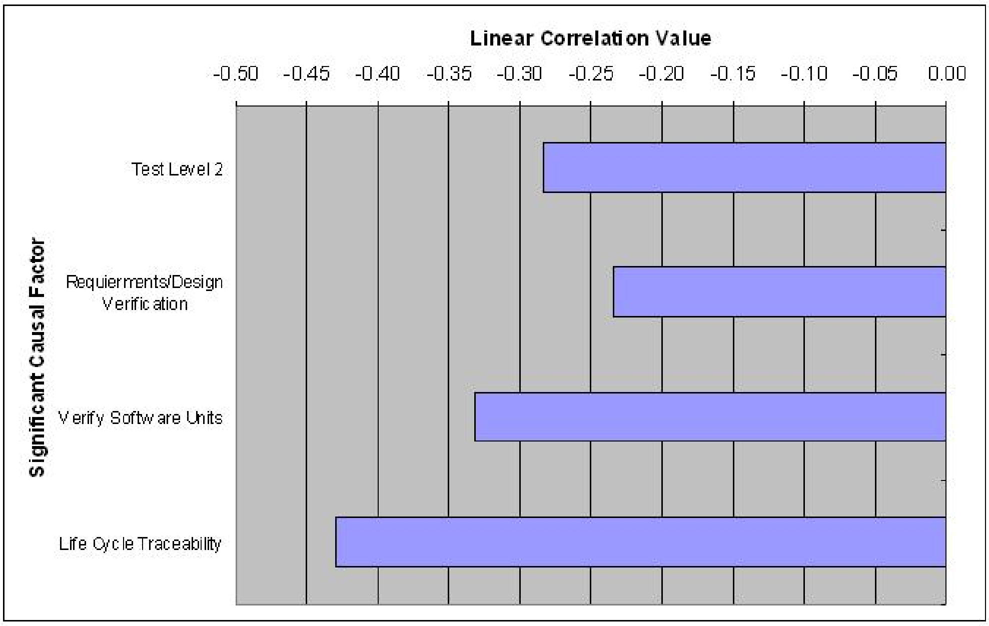

Figure 4.5: Direct Effects of Causal Factors on Functional Adequacy

The factors that significantly influenced Functional Implementation Completeness are shown in Figure 4.6. A total of 10 potential causal factors of software quality were found to be significant when directly correlated to Functional Implementation Completeness. The model inputs that represented development team skill and experience affected Functional Implementation Completeness as expected: the increasing presence of more skilled (Level

4) developers had a positive correlation, and the increasing presence of less skilled (Level 2) developers had a negative correlation. Also, the influence of problem complexity in the form of the Depth of Inheritance Tree had an expected effect, with Functional Implementation Completeness decreasing as the complexity increases.

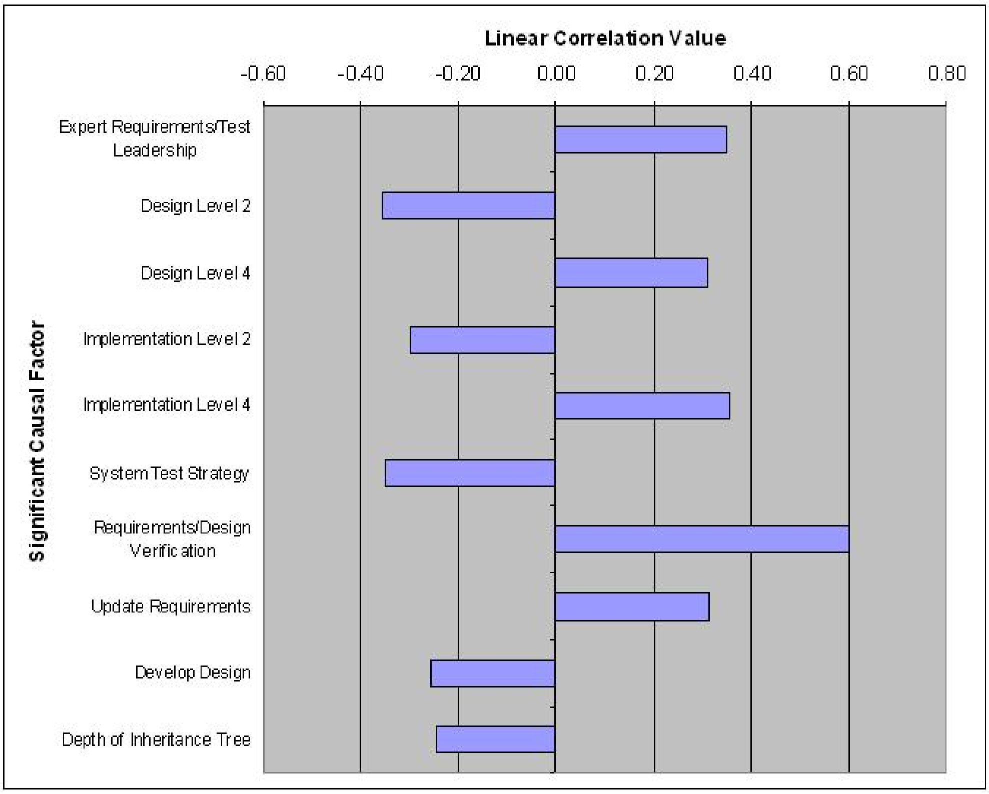

Figure 4.6: Direct Effects of Causal Factors on Functional Implementation Completeness

From a process perspective, most of the process variables had the expected positive correlation with Functional Implementation Completeness. The most notable exception is the Develop Design process. Recall that the Develop Design process is a consolidation of three practices that methodically document and evolve a software design. While intuitively this practice should increase the quality of the software product, in the area of implementation completeness, which addresses coverage of the requirements in the final product, the act of methodical design appears to decrease requirements coverage. One could infer from these results that the process of developing the design actually distracts the development team from completely addressing the requirements. That is, because the focus is the design process and not the design product, the development team loses sight of the goal.

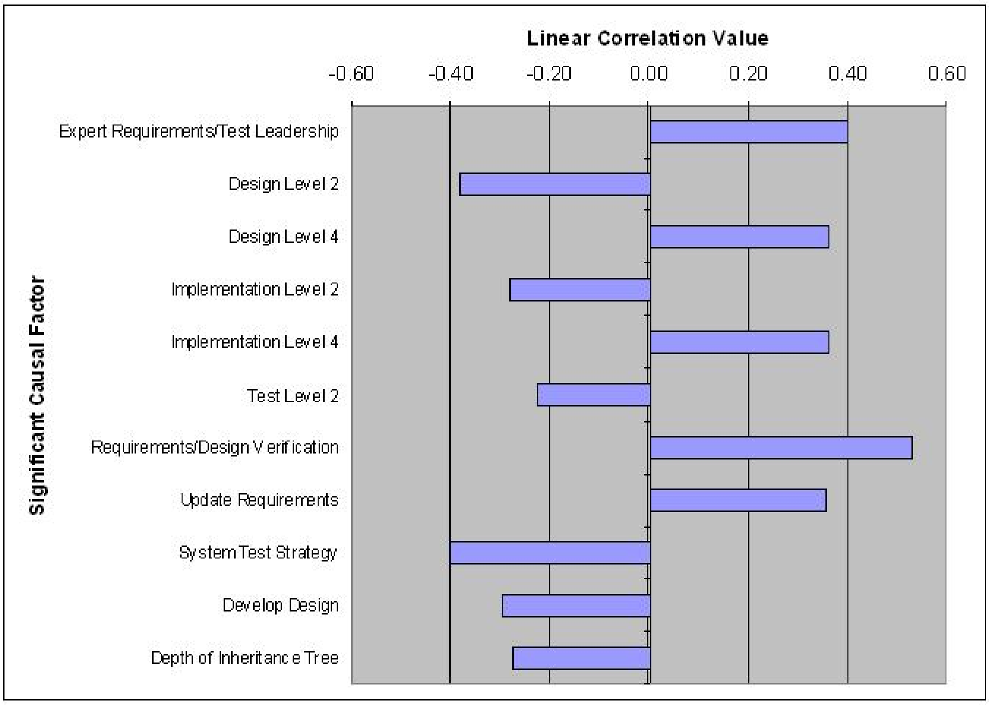

Figure 4.7: Direct Effects of Causal Factors on Functional Implementation Coverage

The effects of the various causal factors of software quality on the Functional Implementation Coverage variable is captured in Figure 4.7. The variables that had a significant impact on this software quality attribute were nearly identical to those that had a significant impact on Functional Implementation Completeness. This is not surprising given the nature of these two variables. While Functional Implementation Completeness addresses the software products coverage of the full set of requirements, Functional Implementation Coverage addresses the correctness of that implementation. Of particular note for this variable is the positive correlation of those processes that represent verification of a work product. The practices of evaluating the requirements and design to verify their correctness are the two most highly influential variables for this software quality attribute.

Functional Specification Stability characterizes a project's ability to resist making changes to the completed specification document. Specification changes have a direct effect on the volatility of downstream artifacts, and often will introduce a lot of rework. Most projects make an effort to be very judicious about the specification changes they allow because of this adverse effect. Figure 4.8 lists the set of model inputs that had a verified correlation with Specification Stability. Not surprisingly, skill expertise has a significant positive impact on this variable. In addition, the Update Requirements process, which intrinsically promotes an iterative approach to development had a significant positive effect.

#### *4.3.1.2 Computational Accuracy*

Computational Accuracy is the sub-characteristic of Functionality that addresses the correct implementation of algorithms. It is primarily focused on quality in terms of accuracy of any calculations implemented in the software, and in any precision of data results. In the data sets, Accuracy and Precision were quality needs in only a subset of the projects, and so their sample sizes are less prolific than those of the Suitability metrics.

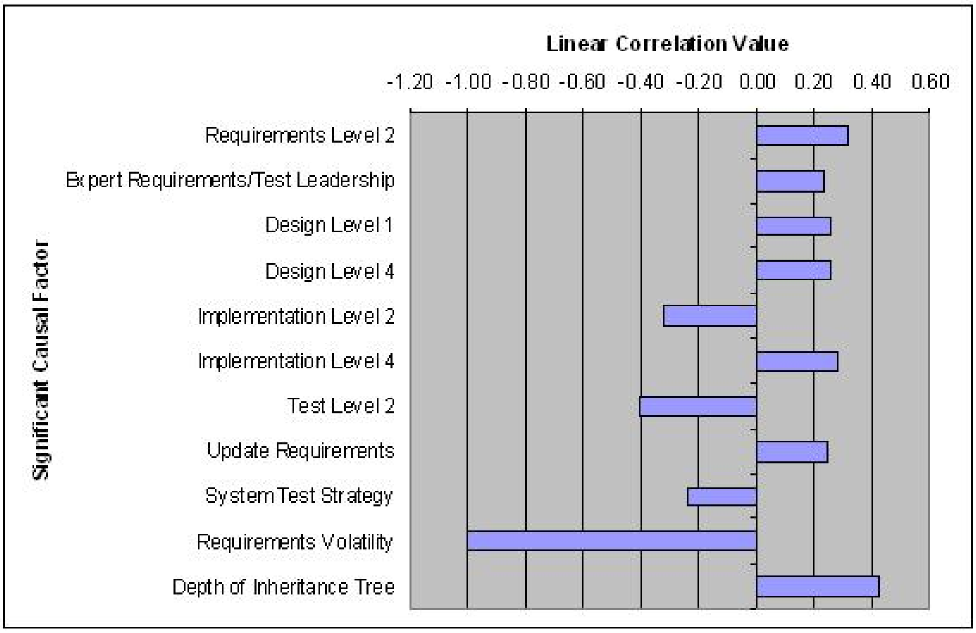

Figure 4.8: Direct Effects of Causal Factors on Functional Specification Stability

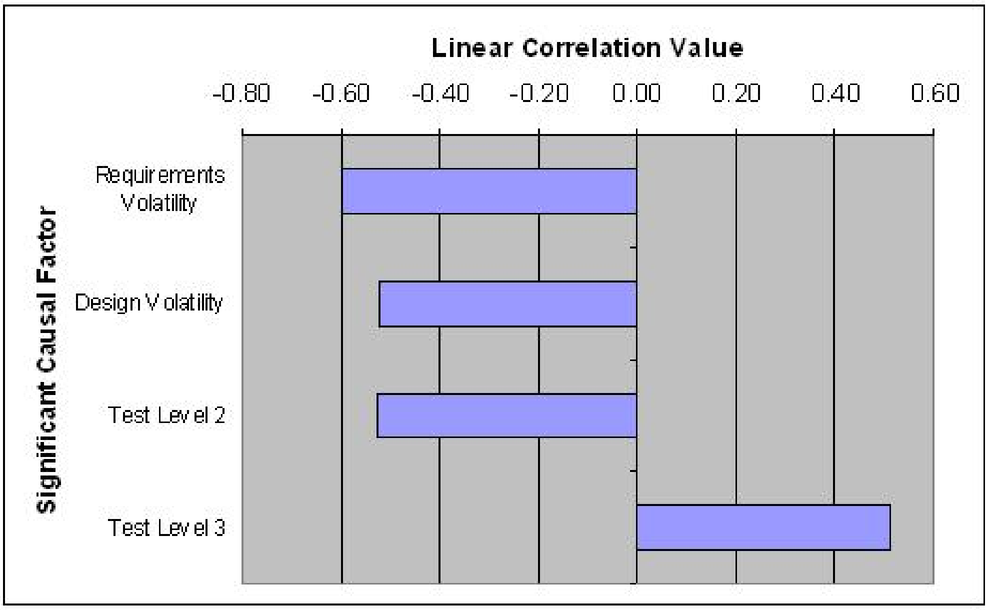

Figure 4.9: Direct Effects of Causal Factors on Accuracy

Figure 4.9 identifies the driving factors that were directly correlated to the measure of Accuracy. Accuracy was influenced negatively by volatility in the requirements and design phase, and also by Test Skill at Level 2. Interestingly, Test Skill at Level 3 provided the only significantly positive effects on Accuracy. This implies that skill and experience in developing, executing and interpreting software tests is necessary to verify that accuracy requirements are met. The effects of requirements and design volatility are cause to suspect that Accuracy requirements and designs that are not decidedly fixed at the time of baseline are difficult to implement correctly. This is reasonable in that custom algorithms implemented in software are typically complex, and thus frequent changes can be detrimental.

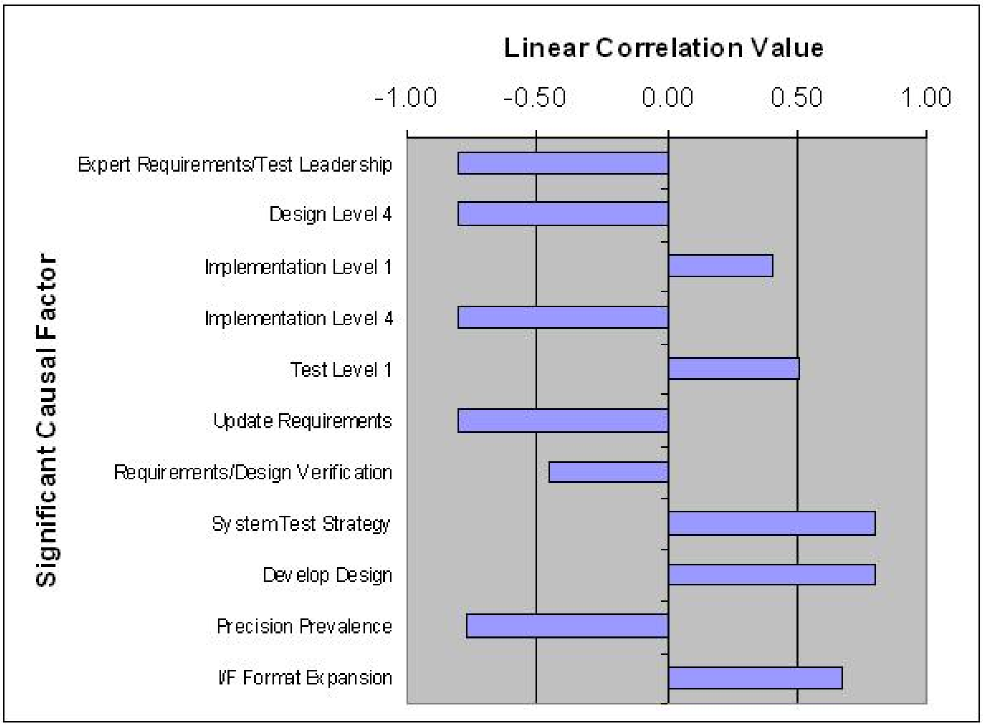

Figure 4.10: Direct Effects of Causal Factors on Precision

The effects of the various causal factors of software quality on the Precision attribute is captured in Figure 4.10. Precision addresses that subset of the requirements that pertain to the precision of data and/or data types within the system. The presented results are curious because they defy the expectations for measures of software quality. For Precision, the increasing presence of less skilled developers in the Implementation and Integration/Test life cycle were positively correlated to high values of Precision. Conversely, the increasing presence of experts in each of the four life cycle phases were negatively correlated to measures of Precision, implying that the presence of experts hurt the likelihood of verifying the implementation of precision requirements. Of the processes, only Develop Validation Criteria, Develop Design and Develop System Tests are positively correlated, while nearly all of the processes that stress verification of work products are negatively correlated.

#### *4.3.1.3 Interoperability*

In the ISO/IEC 9126 Software product quality standard, Interoperability is the quality sub-characteristic that captures software quality as it applies to interfaces. Two variables were selected from the ISO/IEC 9126: Data Exchangeability and Interface Consistency. Data Exchangeability is a measure of the proportion of interface data format requirements that were verified to have been implemented correctly. Interface Consistency measures the proportion of interface communication protocol requirements that were verified to have been implemented correctly.

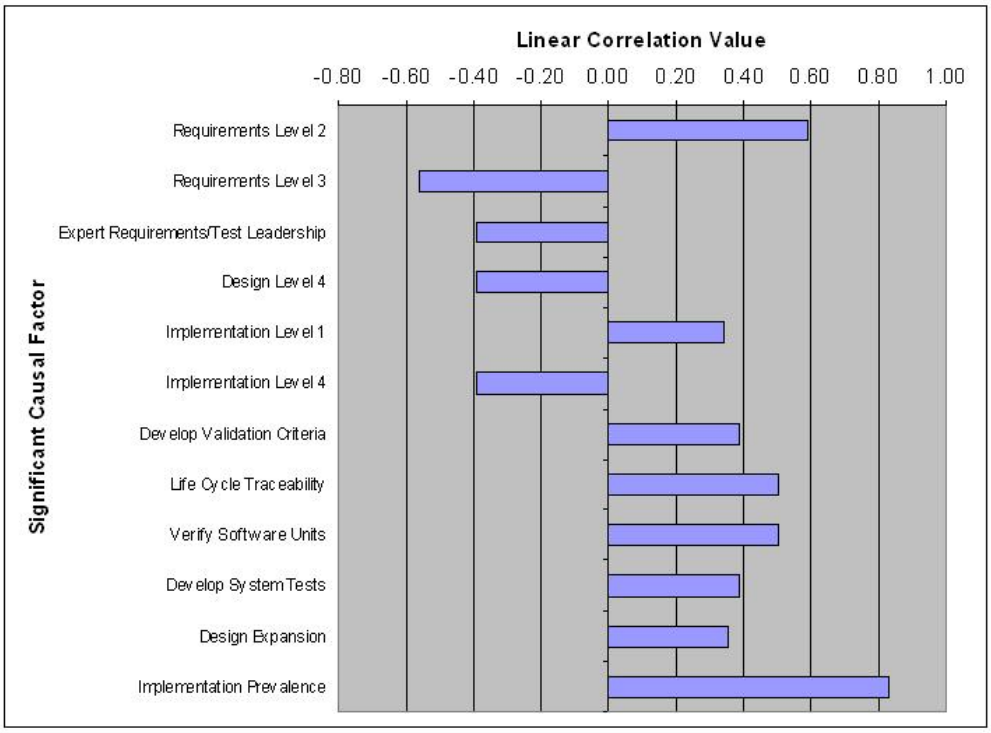

Figure 4.11: Direct Effects of Causal Factors on Data Exchangeability

In Figure 4.11 is listed the set of model input variables that were found to be significantly correlated to the Data Exchangeability attribute. From a Development Team Skills perspective, Data Exchangeability is positively influenced by the presence of less skilled engineers in both the Requirements and Implementation life cycle phases, and negatively influenced by the presence of ”experts” (Skill Level 4) in all of the life cycle phases. This inverse relationship from what would intuitively be expected cannot be explained. The most influential correlation is that of implementation prevalence, which is positively correlated with Data Exchangeability. It is proposed that the increasing presence of a well-defined design greatly simplifies the implementation, and so a higher prevalence of this simplified implementation leads to higher software quality for this attribute. Other positive influences seem to focus on verification and traceability throughout the life cycle. It can be argued that for implementation of interface formats, verifying the correctness of that format, and accounting for all data structures to be implemented are paramount to high quality in Data Exchangeability.

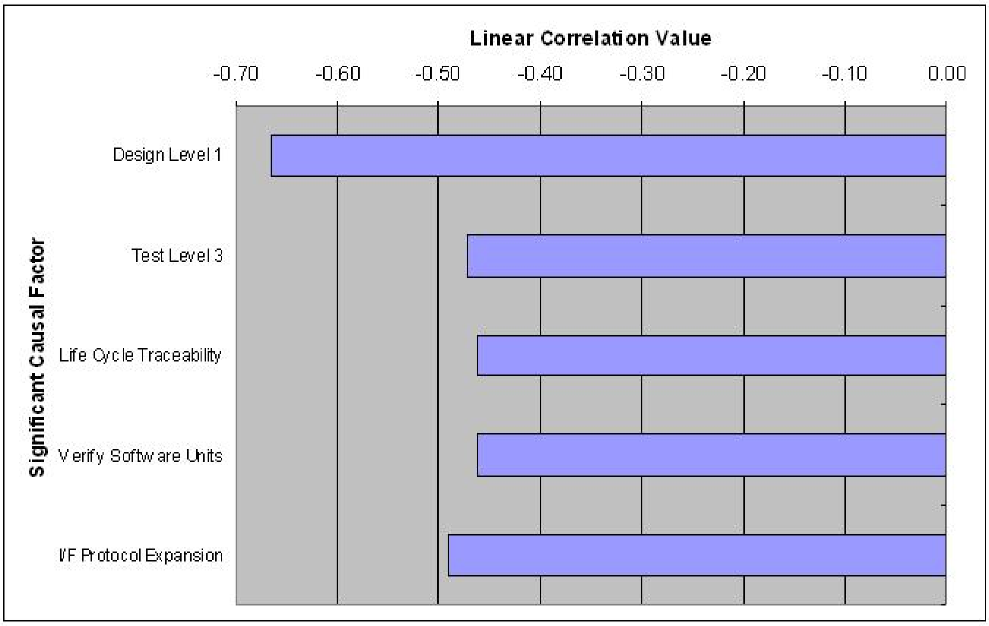

Figure 4.12: Direct Effects of Causal Factors on Interface Consistency

The effects of the various causal factors of software quality on the Interface Consistency quality metric are shown in Figure 4.12. Interface Consistency captures the aspect of interoperability associated with correct implementation of the interface protocols. As with Functional Adequacy, all of the significant correlations to this software quality attribute were found to be negative. The most influential model input was the Design Skill Level 1. This is logical in that the presence of inexperienced designers can certainly negatively affect software quality with respect to the interfaces. The practices that address traceability for each of the requirements, design, and implementation phases were negatively correlated to

Interface Consistency. In terms of problem complexity, increasing values of Interface Protocol Expansion negatively affect this quality variable. Thus, as expected, an increasing proportion of protocols for the project to support increases the complexity of the interface software, and decreases the associated quality.

#### *4.3.1.4 Security*

Security is the aspect of Functionality that addresses how the software controls access to the system. Security is concerned with both access restrictions and data restrictions. Access Auditability accounts for any requirements to create types or levels of access to the software system. Access Controllability addresses any requirements to restrict access to the software system. Data Encryption refers to the need to protect individual data elements from unauthorized access. The three Security quality metrics are represented in a subset of the software projects used for data in this research.

Figure 4.13 shows the effects of the significant causal factors on the Access Auditability quality metric. This aspect of Security covers any requirements that address user access to the system and auditing those accesses. The negative influence of design volatility and interface complexity are expected with respect to any quality attribute. The negative influence of both verification and traceability practices, however, are less easy to explain. In terms of verification, perhaps involving the customer in the evaluation of this particular quality attribute confounds the issue. The intent behind Access Auditability is straightforward:

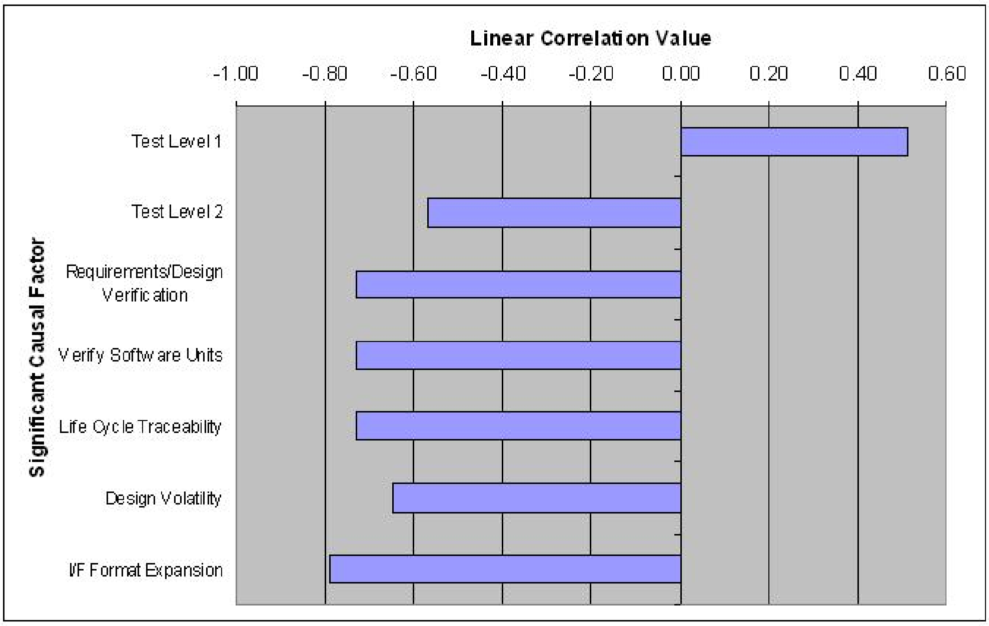

Figure 4.13: Direct Effects of Causal Factors on Access Auditability

what are the customer needs with respect to operator access, and being able to trace that access. However, often users will exaggerate their needs in this quality area. Often, the goal is to maximize the security of the system without regard to what the actual needs are. Only at test time, when a complicated accessibility scheme is determined to be infeasible, is this realized. The user often then settles for their actual needs. In this scenario, it is possible to see how verification may have a negative influence on the Access Auditability quality attribute.

The software quality model inputs that affect the Access Controllability quality variable are identified in Figure 4.14. Access Controllability is positively influenced by verification activities in the requirements and design phase. Specifically, verifying requirements with the

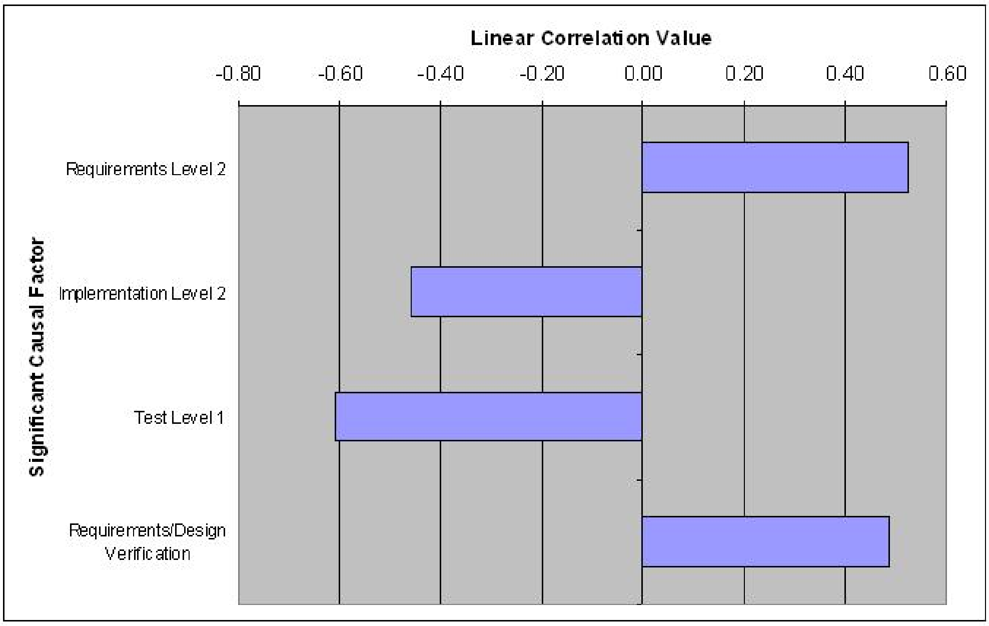

Figure 4.14: Direct Effects of Causal Factors on Access Controllability

customer and verifying the design. Inexperience in implementation and test has the most influential negative effect on implementing software to control access to the system.

The factors that significantly influenced Data Encryption are shown in Figure 4.15. Data Encryption was required in only two software projects in the data set, and so correlation values reflect that small sample size (all correlations are of value 1.0). Data Encryption is almost exclusively affected by Development Team Skill. That is, the increasing presence of skill levels in all life cycle phases is linked to quality in terms of fulfilling Data Encryption requirements. Despite this seeming significant influence, the small sample size of Data Encryption in this research (only two projects identified this as a software quality need) should be cause for concern. The results of this software quality variable are reported here simply

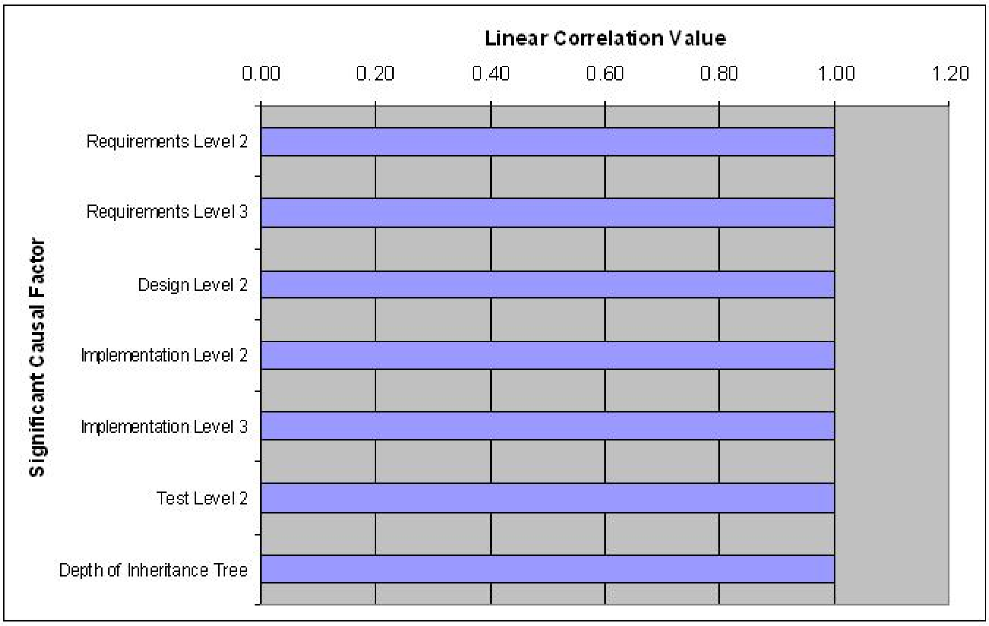

Figure 4.15: Direct Effects of Causal Factors on Data Encryption

to identify the subset of causal factors that was used in the Direct Effects model. However, it may be premature to infer how these factors truly influence Data Encryption quality.

#### *4.3.1.5 Functional Compliance*

Functional Compliance is the portion of Functionality which measures the extent to which the software complied with any levied functional standards. The functional standards may be scoped to an internationally approved standard, or may be organization-specific. Figure 4.16 identifies the factors that were calculated to have a direct effect on Functional Compliance. Only two of the causal factors were significant in their correlation to Functional Compliance, both focused in the implementation life cycle phase.

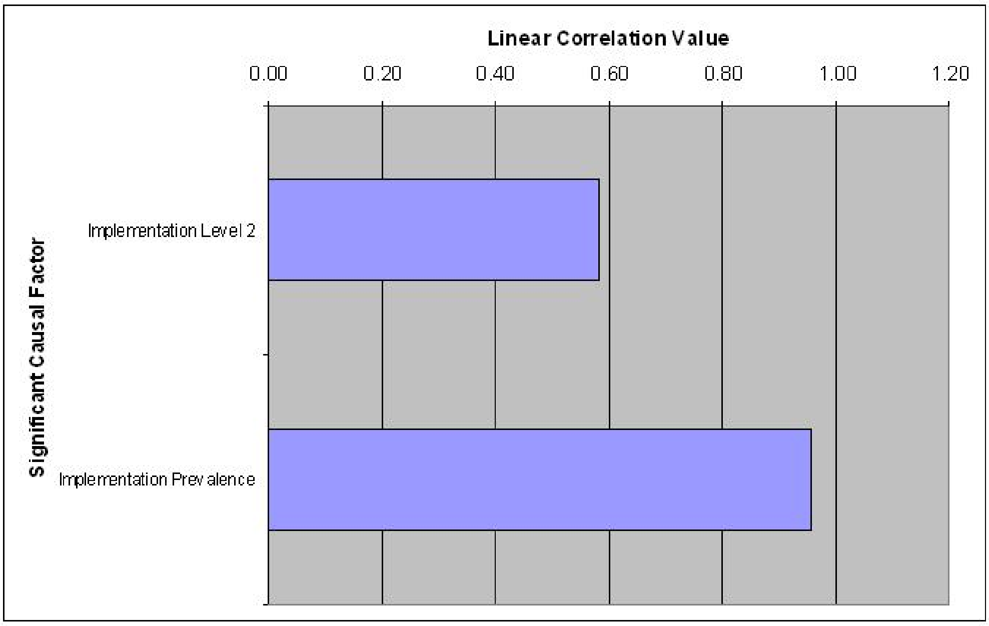

Figure 4.16: Direct Effects of Causal Factors on Functional Compliance

A measure of Implementation Skill (Level 2), and Implementation Prevalence were positively correlated with Functional Compliance. This is surprising as a negative correlation for each would have been more intuitive. Implementation Prevalence, which measures the proportion of the source code impacted by the Functional Compliance requirements, is a measure of complexity and thus it is expected that an increase in prevalence will cause a decrease in the quality associated with Functional Compliance. The positive correlation is an indication that the implementation of functional standards is less complex in that the behavior is already defined. That is, the larger prevalence of more standardized functional behavior actually decreases the complexity of the design because it propagates recognized and well understood software operations. The simplification of the software through widespread implementation of standardized behavior may also serve to explain why those less skilled, Implementation Skill Level 2, are significantly correlated to higher quality for this attribute.

### *4.3.2 Analysis of the Reliability Software Quality Attributes*

Reliability is the portion of the ISO/IEC 9126 standard that addresses the software product's ability to handle failures, operator errors, and recover from other unexpected events. Reliability is typically expressed in software requirements as off-nominal conditions or alternative use case scenarios, and represent a subset of the actual software requirements (if they are captured at all). The focus for the measures of reliability is to make the developed software available, even in the face of failures. The categories used to measures Reliability include Failure Avoidance, Incorrect Operation Avoidance, Restorability, and Reliability Compliance. In this section, these measures are discussed in terms of any discovered direct effect relationships with those variables that are drivers of software quality.

The effects of the various causal factors of software quality on the Failure Avoidance attribute are captured in Figure 4.17. Failure Avoidance is the subset of the requirements that specifically address identified failure scenarios, and requires the software to recover from those scenarios. Testing skill was an influence on Failure Avoidance, but curiously the nature of the relationship was not as expected: the presence of more skilled testers (Level 3) was negatively correlated and the the presence of less skilled testers (Level 1) was positively correlated. Similarly, as the complexity of the developed software, characterized by the

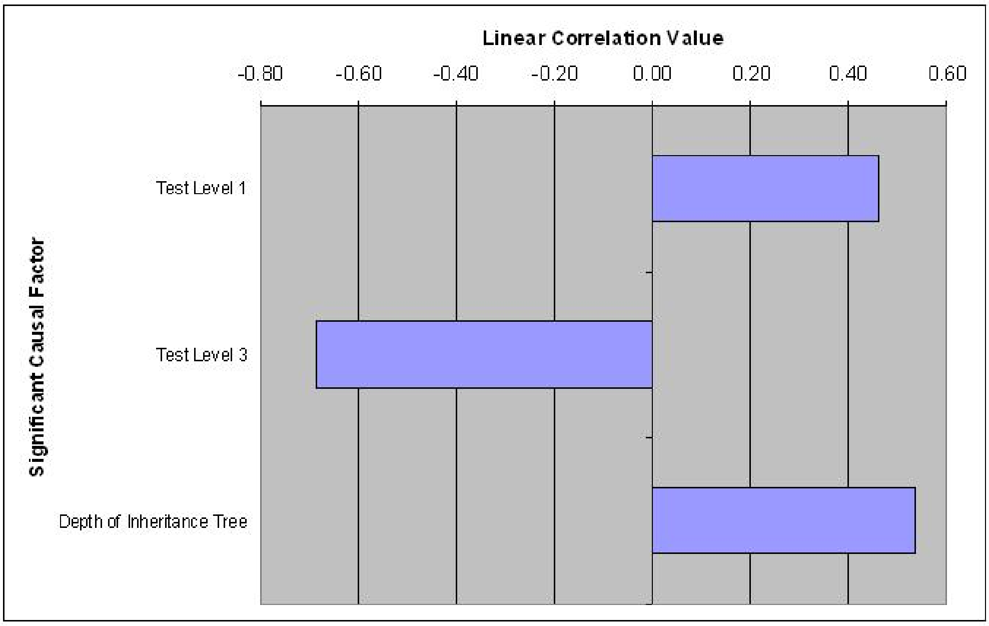

Figure 4.17: Direct Effects of Causal Factors on Failure Avoidance

depth of the inheritance tree, increased, the percentage of Failure Avoidance requirements that were verified to be correctly implemented also increased.

Incorrect Operation Avoidance is the set of requirements that address the ability of the system to remain available in the presence of incorrect operation. The model inputs that had a significant impact on this quality attribute are shown in Figure 4.18. The majority of factors that had an effect on Incorrect Operation Avoidance were Skill/Experience variables. All of these were correlated as expected with higher skill levels positively correlated and lower skill levels negatively correlated. All significant process variables, however, are negatively correlated. This implies that instrumenting the software with functions to gracefully handle an operator's misuse of the system is impaired by verification and traceability processes.

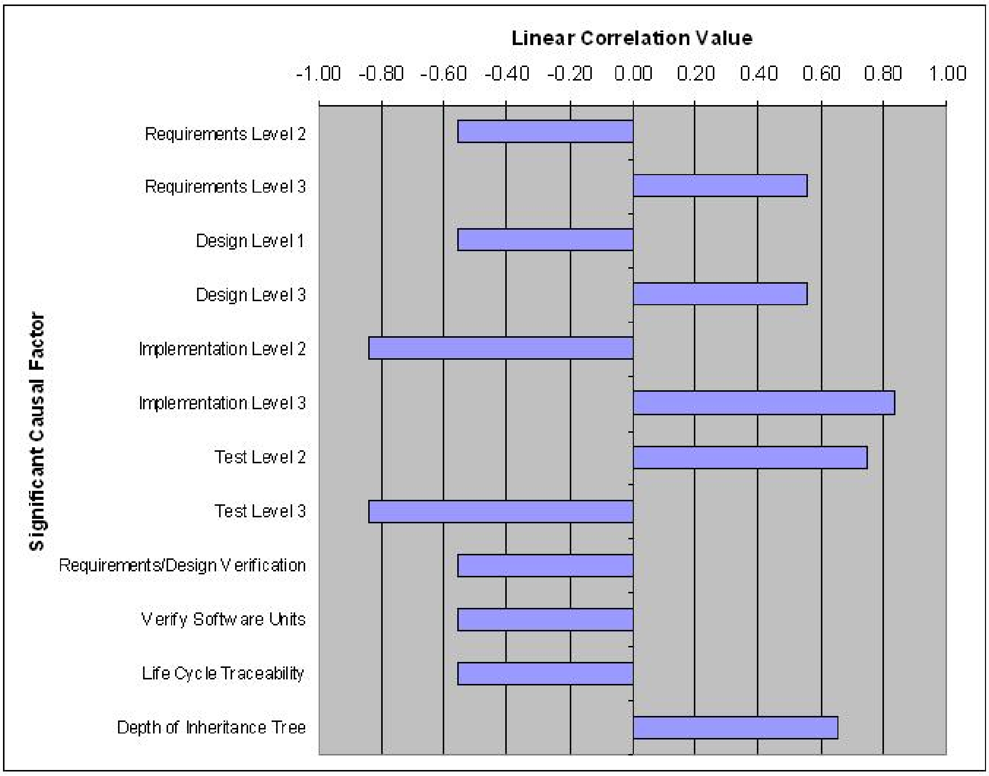

Figure 4.18: Direct Effects of Causal Factors on Incorrect Operation Avoidance

Restorability is the ISO/IEC 9126 quality attribute that captures the ability of the system to restore itself to a known state after a failure event. Seven of the projects in the acquired software engineering data had quality needs in Restorability. Figure 4.19 displays the set of skill, process, and complexity factors that had an influence on this quality variable. In terms of causal factors, nearly all of the inputs affecting Restorability had correlation values that were expected. Higher skill levels were positively correlated with Restorability, and lower skill levels were negatively correlated. Design Expansion, a measure of complexity, was negatively correlated. Process variables were positively correlated except the Develop Design variable, which was found to be negatively correlated with Restorability.

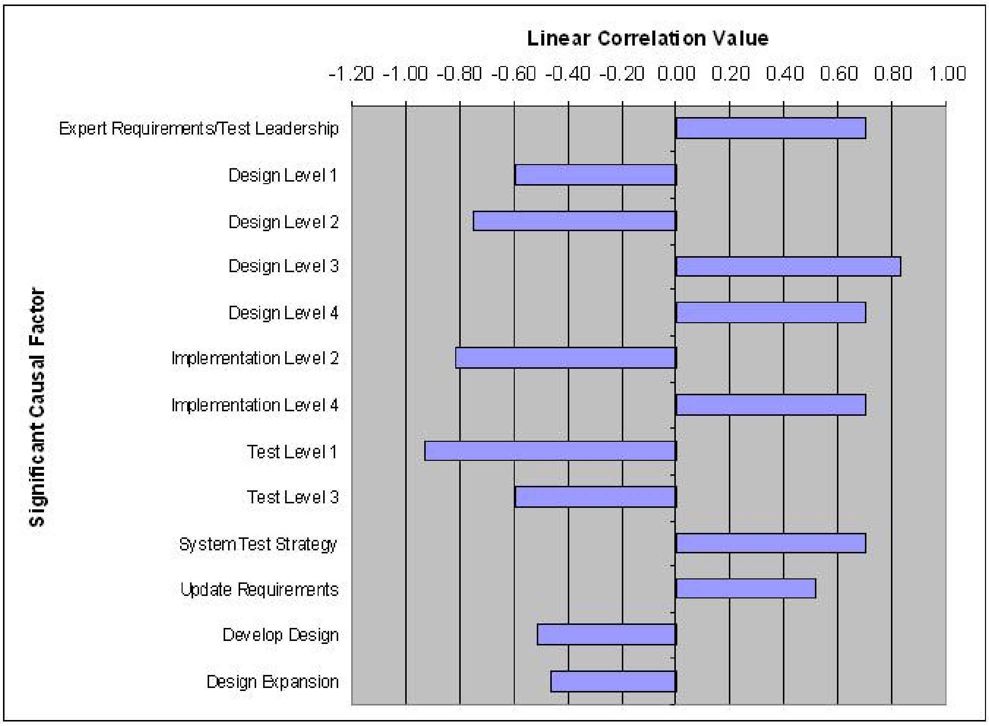

Figure 4.19: Direct Effects of Causal Factors on Restorability

Reliability Compliance is not included in the Direct Effects analysis. This software quality variable is addressed in only three of the projects used in this research. Similar to the situation encountered in the development of the Refined Model structure, none of the causal factors used as model inputs could be correlated to Reliability Compliance to the desired degree.

### *4.3.3 Analysis of the Efficiency Software Quality Attributes*

Efficiency refers to the run-time performance of the software quality product. Resource utilization and software behavior with respect to time are all aspects covered by Efficiency in the ISO/IEC 9126 Software product quality standard. I/O Utilization, Efficiency Compliance, and Time Behavior are all quality sub-characteristics that were identified as needs in a subset of the projects used in this research.

Input/Output (I/O) Utilization refers to those requirements that express performance needs in the software with respect to the physical machine's I/O devices. Typically, these requirements express a performance target to be met, or a performance budget to operate within. Figure 4.20 shows the eight model inputs that had a verified correlation with I/O Utilization. Due to the small sample size of I/O Utilization variables (only two projects had this quality need), the set of causal factors that are correlated have a correlation coefficient of 1.0. Based on the small sample size, it would be premature to make any inferences about the factors influencing I/O Utilization. The results are shown here for completeness, but this quality attribute is not considered in the analysis of the most significant influential factors on software quality (presented in Section 4.3.7).

Efficiency Compliance provides a means to capture any performance or capacity quality needs that were identified by a project, but cannot be categorized into any of the other Efficiency quality sub-characteristics. Three of the projects in this research had these types of quality needs. Figure 4.21 lists the subset of causal factors that were found to be significantly correlated with the Efficiency Compliance model variable. As with I/O Utilization, the small sample size makes it difficult to make inferences about these influences. However, it seems clear that skill and problem complexity are driving factors. Skill variables from all four life cycle phases were identified. In addition, requirements volatility was identified as a driver

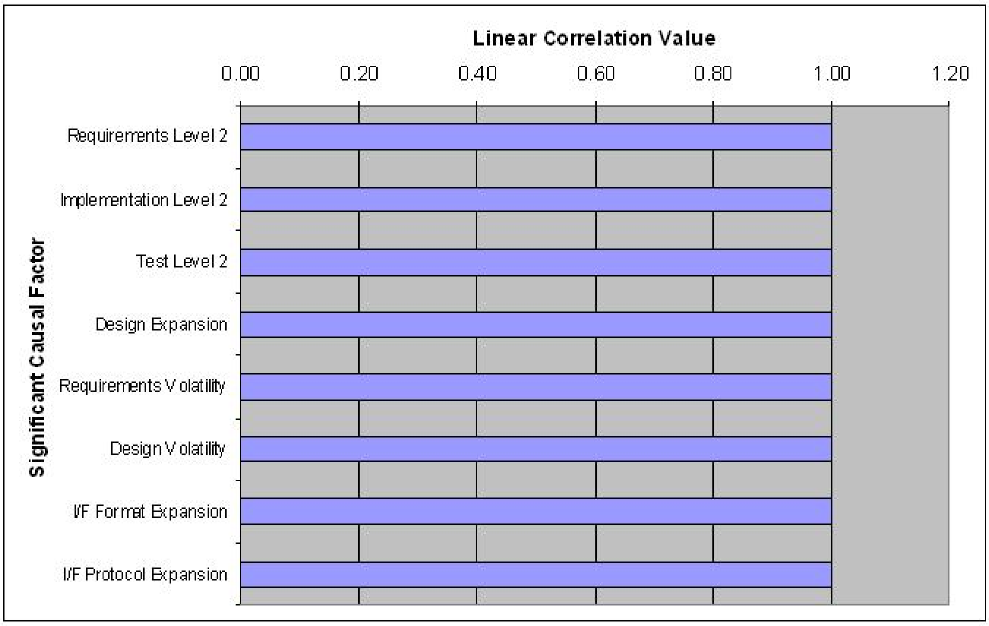

Figure 4.20: Direct Effects of Causal Factors on I/O Utilization

for Efficiency Compliance. This is reasonable as performance and capacity requirements are typically solidified as a statement in the specification at requirements time, but with actual performance and capacity values listed as "To Be Determined". It is often later in the life cycle when those values are more concretely specified. Thus, the positive correlation of Efficiency Compliance with Requirements Volatility is logical. Another interesting relationship is that of Efficiency Compliance with the interface complexity measures. One inference that can be made is that this quality category is typically associated with the performance of the software in terms of relating to its interfaces as opposed to internal operations.

Time Behavior is not included in the Direct Effects analysis because no direct correlations could be verified between it and the model inputs. This software quality variable is addressed in only three of the projects used in this research. Similar to the situation encountered in

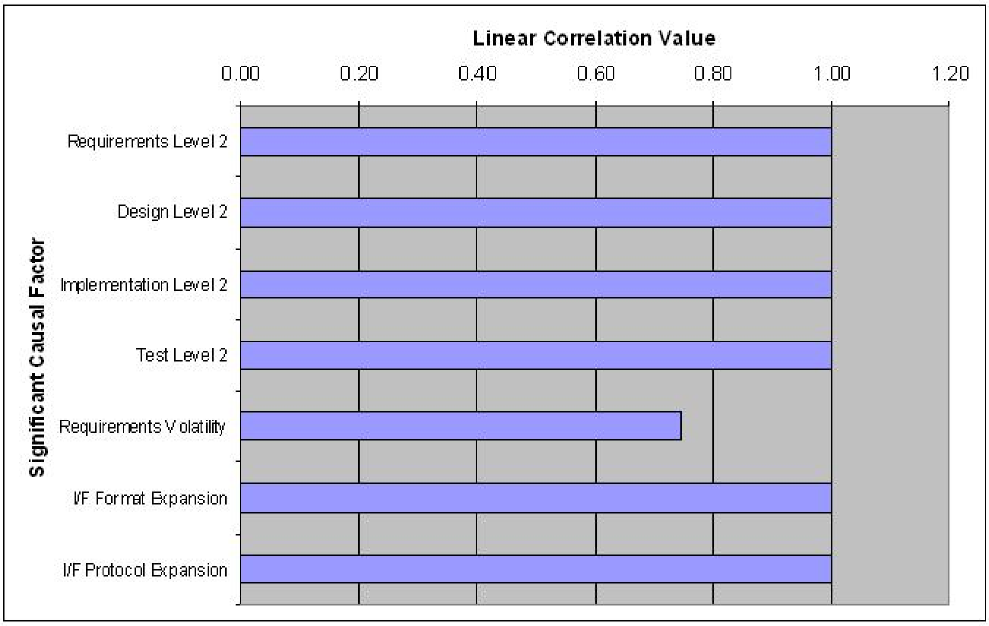

Figure 4.21: Direct Effects of Causal Factors on Efficiency Compliance

the development of the Refined Model structure, none of the causal factors used as model inputs could be correlated to Time Behavior to the desired degree.

### *4.3.4 Analysis of the Usability Software Quality Attributes*

Usability is the portion of the ISO/IEC 9126 Software product quality standard that addresses the way in which the software interacts with the operator. Quality characteristics such as the ability to cancel and undo user operations, accommodations in the user interface for those with physical disabilities, and the ability to communicate the operational status of the software system are all covered in the Usability partition of the standard. In this research, three of the software quality metrics that characterize Usability were quality needs of the projects that comprised the set of software engineering data: User Cancellability, Operational Status Monitoring, and Usability Compliance.

User Cancellability covers a very specific aspect of operator interaction: the ability for a user to cancel an operation that has previously been initiated. From an implementation standpoint, this feature often involves tracking how an operator-initiated operation has changed the state of the system, and returning the system to the original state in the event that the operation has been canceled. Only one model input variable was correlated to User Cancellability. The Design Level 2 variable, or marginal design skill, was positively associated with User Cancellability. This lone correlation is not easily explained. As with a couple of other variables in this study, the small sample size (three projects) makes it premature to make inferences about cause and effect. However, this study has identified that a verified correlation is required for the Direct Effects model structure. Design Level 2 is the only variable that passed this criteria, and so was used in the Bayesian Direct Effects model, the Least Squares Regression model, and the Neural Network model (see Sections 4.4 and 4.5).

Figure 4.22 depicts the subset of software quality causal factors that were found to be significantly correlated to Operation Status Monitoring. This attribute encompasses those software quality needs that pertain to the availability of information to the user on the operational status of the software system. Seventeen of the causal factors were verified to correlate to Status Monitoring, and their coefficients were inline with expectations. From a skill/experience perspective, higher skill levels (Level 3 and Level 4) in all life cycle phases were positively correlated with this quality attribute, and lower skill levels were negatively correlated. In the area of problem complexity, design expansion had the strongest negative effect. This is logical as design expansion captures the relative size of the software system, and a larger system would intuitively make verifying requirements more challenging. From a process perspective, the practices of verification were positively correlated.

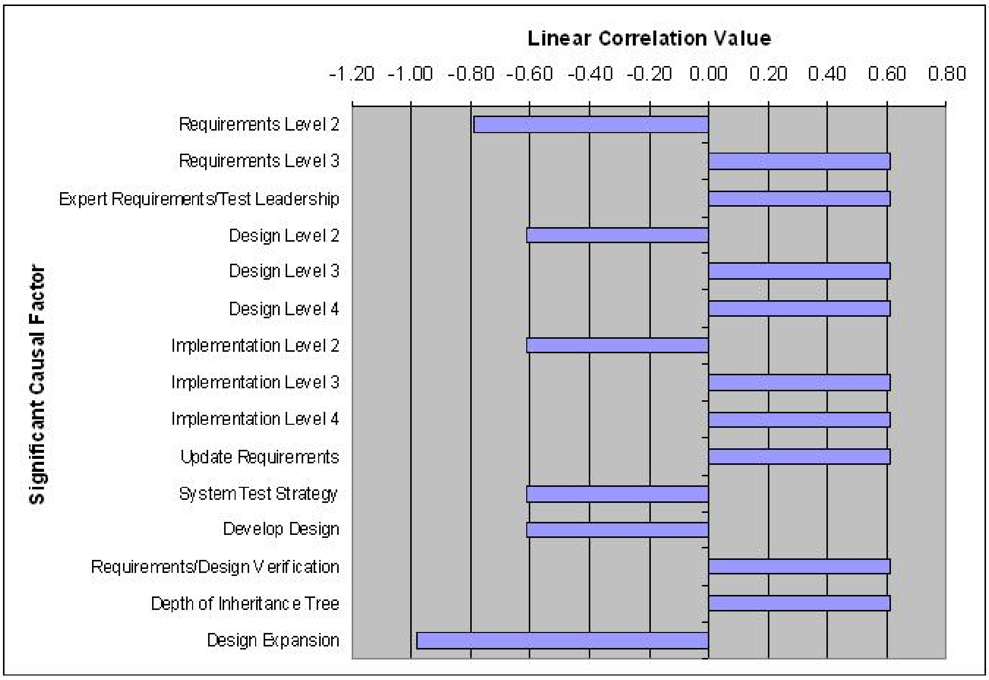

Figure 4.22: Direct Effects of Causal Factors on Operation Status Monitoring

Usability Compliance captures the set of usability requirements that are not easily categorized in other sub-characteristics. In addition, Usability Compliance involves requirements to comply with a specified usability standard or convention. Seven projects from the set of software engineering data identified needs for Usability Compliance. The results of the correlation analysis for this variables is captured in Figure 4.23. The factors that influence Usability Compliance were very similar to those that affected Operational Status Monitoring. Skill and complexity measures were correlated as expected. One additional factor was the negative influence of Traceability throughout the life cycle.

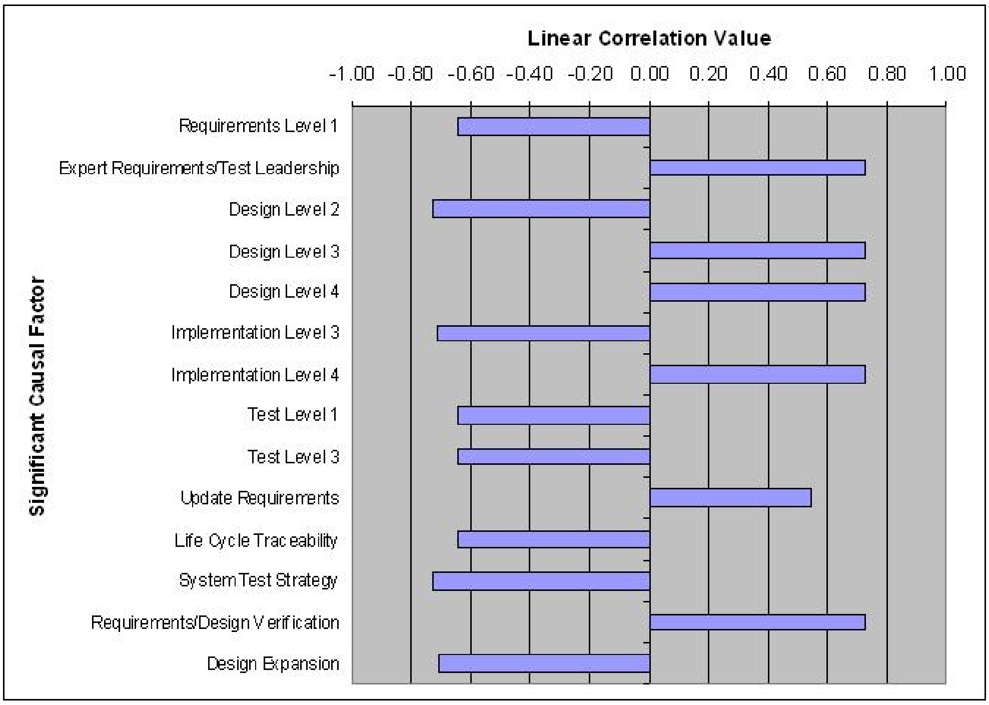

Figure 4.23: Direct Effects of Causal Factors on Usability Compliance

### *4.3.5 Analysis of the Maintainability Software Quality Attributes*

The Maintainability software quality category encompasses those features that assist an operator in troubleshooting the system in the presence of an unexplained event. Activity recording and diagnostic requirements are all elements of Maintainability. For this research, only the Activity Recording software quality variable is modeled from the Maintainability category.

Activity Recording is the quality sub-characteristic that addresses the ability of the software to log significant system events that occur. The purpose of this is primarily as a troubleshooting aid: it allows an operator to reconstruct the events that occurred, possibly in time order. Figure 4.24 shows the software quality factors that affected Activity Recording. Design skill and experience was the only factor that positively influenced this quality attribute. Conversely, volatility of that design had the most significant negative effect on Activity Recording. One interesting revelation is the negative influence of the relevant interfaces on Activity Recording. This could be indicative of situations where event logging with respect to interfaces complicates the software system. That is, if the software system is responsible for recording the events that happen across interfaces as well as internal events, it is reasonable to see how the prevalence of interface functionality in the system would negatively influence the ability of the development team to meet its Activity Recording requirements.

### *4.3.6 Analysis of the Portability Software Quality Attributes*

Portability is a category of quality within the ISO/IEC 9126 Software product quality model that addresses the ability of the software product to operate in conjunction with other software packages, and across operating system and hardware boundaries. The intent of this section is to identify and analyze the elements of development team skill, process maturity, and problem complexity that have a significant influence on the software quality variables associated with Portability.

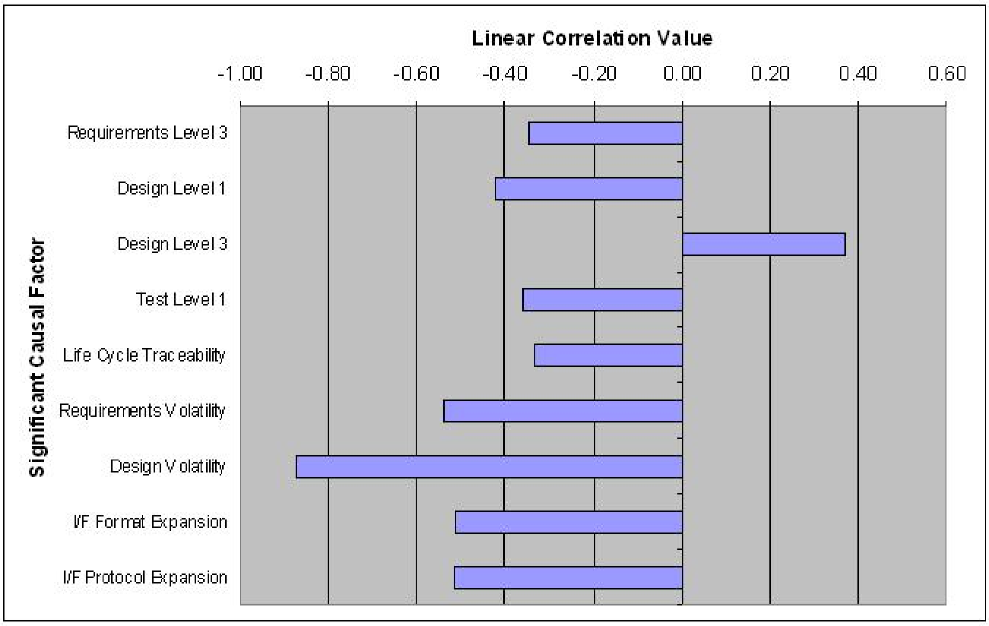

Figure 4.24: Direct Effects of Causal Factors on Activity Recording

Software Operability is the quality attribute that captures the software product's portability across software platforms. This is usually manifested in the form of requirements that specify a set of operating systems on which the software under development must perform. Figure 4.25 shows the model inputs that have a significant effect on Software Operability. The two process inputs and the complexity input affect Software Operability as expected. That is, the presence of practices to trace artifacts through the life cycle and verify the coded software had a positive influence, and the design expansion had an expected negative influence.

Hardware Operability is the operation of the software product across different hardware platforms. This quality attribute is the traditional concept that is associated with portability. Figure 4.26 presents the set of causal factors that significantly influence Hardware

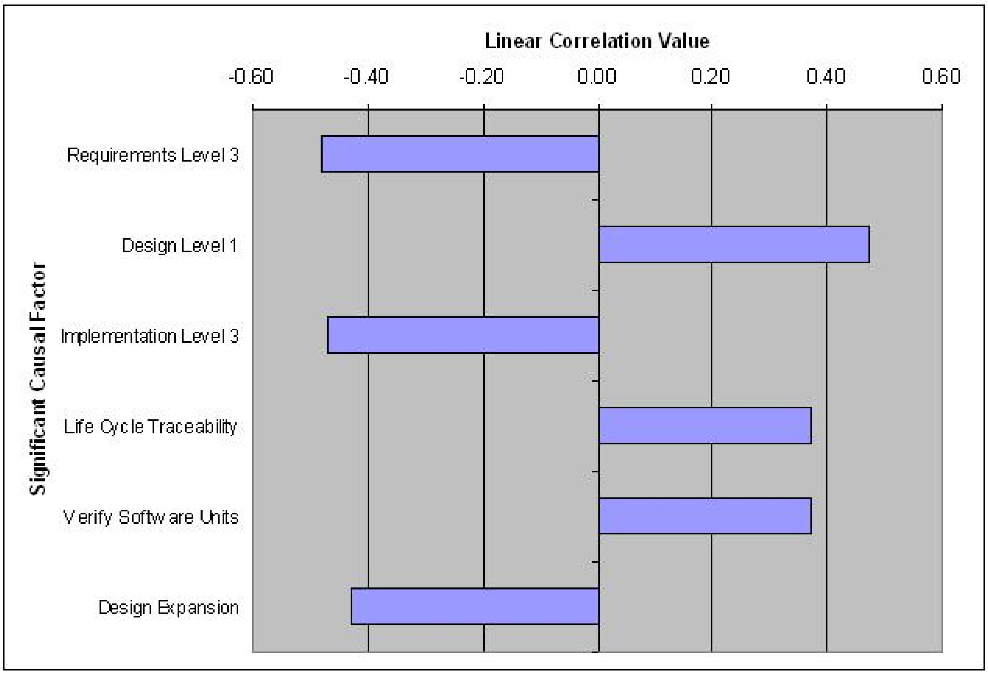

Figure 4.25: Direct Effects of Causal Factors on Software Operability

Operability. Volatility, in both the Requirements and Design Phase, had a significant negative effect. Increasing numbers of requirements and design changes are expected to have a negative effect on any software quality variable.

The skill variables did not perform as expected in relating to either Software Operability or Hardware Operability. Inexperience in software design had a positive effect on this quality variable, and experience in both software design and requirements development had a negative effect. A negative effect means that the presence of more skilled engineers in the areas of Requirements and Design was related to less Software Operability requirements being verified to function correctly. One possible explanation for this phenomenon is overconfidence on the part of more skilled engineers. Non-functional requirements, and particularly those

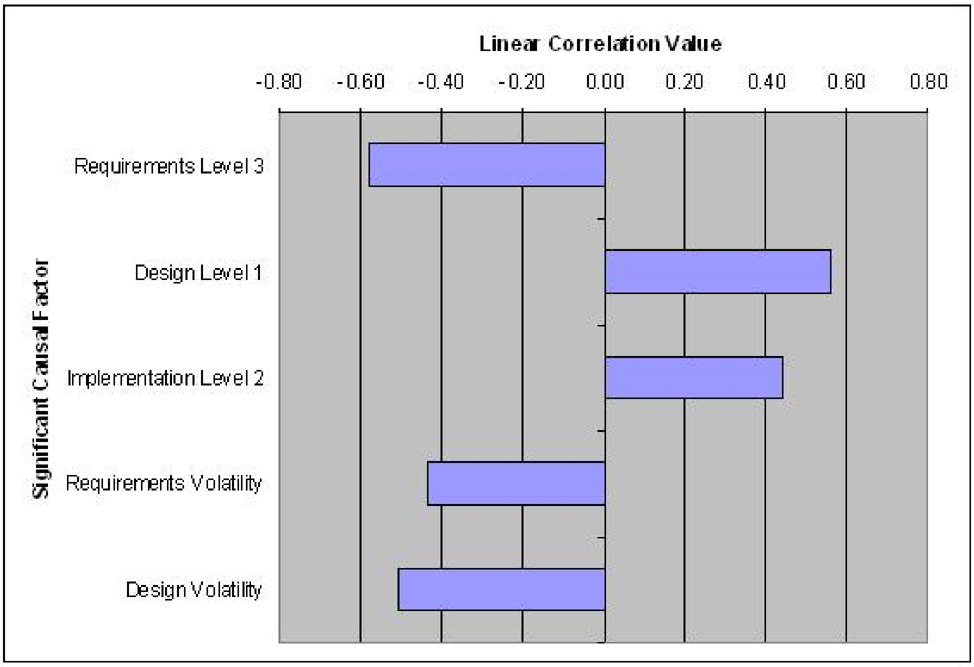

Figure 4.26: Direct Effects of Causal Factors on Hardware Operability

that require little analysis to understand and communicate, typically are overlooked during requirements and design, and are left to be addressed in detail in the implementation. The measures of Portability fall into this category. Requiring a software product to function across a set of operating systems or hardware platforms is easily understood by both the software development team, and the customers. This type of requirement is easy to overlook and leave for implementation, especially if the engineering team is experienced. A novice engineer, conversely, analyzes these types of requirements in full, and does not have the overconfidence to dismiss its potential impact.

### *4.3.7 Analysis of Most Significant Causal Factors*

The analysis of the direct correlations between the various input values and measures of software quality addressed in Sections 4.3.1 through 4.3.6 calls into question the set of inputs that has the most significant impact on software quality. This has many practical implications. An organization that is interested in improving their software development processes would be interested in knowing which of those processes will have the largest return on investment if implemented. Identifying the most significant model inputs can be an important contribution, and can provide skill, process, and complexity areas for an organization to focus on in order to improve their software product quality.

The identification of the most significant model inputs will be partitioned into two areas: identifying those model inputs that have the most positive effect on the software quality variables, and those model inputs that have the most negative effect on the software quality variables. The method of analysis will be to present the significance information in terms of software quality variables, and then by average significance. That is, the information of interest is the number of different software quality variables that the input variable affected, and the average correlation over that number of variables. The source of the information is the Direct Effects analysis performed in the preceding subsections. Recall that Reliability Compliance and Time Behavior were not directly correlated to any model inputs, and so will be excluded from this analysis. In addition, those software quality variables with sample sizes less than three will be excluded, as several inputs correlate to those variables with a coefficient of 1.0, which skews the results.

Table 4.29 presents the model inputs that positively affected 5 or more software quality variables, and their associated average correlation coefficients. These are considered to have the most significant impact on the spectrum of software quality. Of particular note is the presence of skill and experience factors in Table 4.29. The presence of experts, or Skill Level 4, for each of the life cycle phases (recall Requirements and Test Skill Level 4 values are combined in the Leadership variable) is a strong indication of the influence that expertise has on software product quality. From a process perspective, the Life Cycle verification activities such as evaluating requirements with the customer and verifying the design has the most significant impact. This is logical as verification of artifacts is the best way to get an indication of whether the project is on track to fill the customer needs.

The Update Requirements process having a significant influence on overall software product quality is an interesting revelation. Of all the ISO/IEC 15504 Software engineering practices, this is the only one that implies a certain type of software life cycle. Update Requirements addresses the case where a project maintains the requirements specification as part of the transition between iterations in an incremental life cycle model. Thus, the presence of this practice as a significant positive contributor to software quality is an indirect endorsement of incremental development.

The presence of the Depth of Inheritance Tree in Table 4.29 was unexpected. Prior software quality research efforts have identified DIT as a measure of design complexity and have associated this measure with the number of faults or changes in a given software module. The results here indicate an opposite effect, that an increasing DIT is correlated with higher quality values for 4 of the software quality variables modeled. Reconsidering this effect, it seems reasonable. A larger DIT indicates more inheritance in the developed source code. While this does introduce complexity in terms of inheritance, it reduces the size of the code base in that the same functionality would have been implemented in parallel across multiple modules. Thus, the results found here seem more logical in terms of how inheritance is used on a software project. In addition, this finding provides an indirect endorsement of the effect of the object-oriented paradigm on the quality of software products.

Table 4.29: Identification of Most Significant Positive Causal Factors

| Model<br>Causal Factor | No. Quality<br>Variables Affected | Average<br>Significance |
|---|---|---|
| Implementation Level 4 | 6 | 0.507 |
| Expert Reqs/Test Leadshp | 6 | 0.505 |
| Design Level 4 | 6 | 0.496 |
| Update Requirements | 6 | 0.431 |
| Design Level 3 | 5 | 0.619 |
| Life Cycle Verification | 5 | 0.593 |
| Depth of Inheritance Tree | 4 | 0.558 |

Table 4.30 lists the causal factors that have the most negative influence on software quality variables. As with the positive causal factors, the most noticeable feature of this table is the presence of Skill/Experience model inputs. Just as skill and experience are correlated positively with quality, lack of skill and experience are correlated negatively with quality. For the Requirements and Integration/Test life cycle phases, this is particularly

concerning as all skill levels below "expert" appear to have a negative effect on software quality.

Not surprisingly, volatility in both the requirements and design has a significant effect on quality. This supports the findings of other software quality research which identifies volatility as a negative influence on quality. Design expansion, which is a normalized measure of the size of the software project also negatively affects quality as expected.

The process variables that negatively affect software quality are difficult to understand as their influence is not intuitive. System Test Strategy, which incorporates the practices of developing both a validation criteria and a set of system-level tests, does not seem likely to have a negative influence on software quality. Likewise, the processes associated with developing a design, in which design approaches are documented prior to implementation, should not negatively influence software quality. Verification activities appear on both lists - they are one of the most significant positive and negative influences. The cause for this is not clear. One possibility is that these projects made process their focus, instead of product. That is, perhaps these software development efforts concentrated too heavily on meeting the demands of the process (documentation standards for design, system test procedures, verification activities, etc), and focused too little on satisfying the quality needs required in the actual software product. One could infer that for these small-scale projects, the process was infeasible for the complexity of the software product and the time allotted for completion. This isn't an argument against process, but rather an indication that the impact of the selected processes must be well understood for them to be feasibly performed in a software development effort.

Table 4.30: Identification of Most Significant Negative Causal Factors

| Model<br>Causal Factor | No. Quality<br>Variables Affected | Average<br>Significance |
|---|---|---|
| Implementation Level 2 | 7 | -0.518 |
| Life Cycle Traceability | 6 | -0.526 |
| Test Level 3 | 5 | -0.648 |
| Design Level 2 | 5 | -0.565 |
| Design Level 1 | 5 | -0.501 |
| System Test Strategy | 5 | -0.466 |
| Test Level 2 | 5 | -0.402 |
| Design Expansion | 4 | -0.645 |
| Requirements Volatility | 4 | -0.643 |
| Design Volatility | 4 | -0.636 |
| Test Level 1 | 4 | -0.634 |
| Life Cycle Verification | 4 | -0.494 |
| Requirements Level 3 | 4 | -0.491 |
| Develop Design | 4 | -0.419 |

## 4.4 Comparison of Bayesian Model Structures

In Chapter 3, an intuitive model for software quality was proposed. This model was based on experience, and justified through a logical analysis of what factors in the development of software are expected to be influences on the quality of the product. In Sections 4.1-4.2, the intuitive model structure was analyzed in terms of correlations within the model structure. The result was a model that kept the general form of the original structure, but was refined in that only those cause-effect relationships justified through empirical correlations were retained. Finally, Section 4.3 analyzed the direct effects of the set of measures used as model inputs to the set of software quality measures. This analysis examined the potential for a much simpler model structure in which there are no intermediate nodes in the structure of the model, simply a direct relationship between inputs and outputs.

This section compares the three approaches to modeling software quality: Intuitive model, Refined model, and Direct Effects model. In all three cases, the mechanism for modeling software quality is the Bayesian Belief Network, but the approach to structuring the model is different. The purpose of this section is to determine which of the three approaches characterizes software quality more accurately, and identify any patterns that may exist between the quality measures being evaluated, and the structural type being compared.

Sections 3.3.2.3 and 3.3.2.4 describe the methods to be used to compare the Bayesian model structures. Accuracy of Fit will be determined by verifying the Equality of Means and Equality of Variances between modeled and actual values of software product quality. Predictive Validity will be compared in terms of the measure of Average Relative Error for the predicted values of software quality versus the actual values. In addition to these criteria, this section will also analyze the Belief associated with predicted values. Bayesian Belief Networks are unique as a forecasting model in that they provide a quantified level of confidence, called Belief, in addition to a value of prediction. Thus, a prediction using a Bayesian Belief network is always accompanied by a measure of confidence in the prediction. The Belief value will be considered as a criteria for weighting the results of the forecasts made using each of the three Bayesian structures.

### *4.4.1 Accuracy of Fit Analysis*

The Accuracy of Fit for the three Bayesian structures involves determining whether the structure models each software quality variable such that modeled values are comparable to actual values. The technique for determining this is to verify that the mean and variance of the modeled values are equivalent to the mean and variance of the actual values, as described in Section 3.3.2.3. Figure 4.27 compares the Equality of Means results for each of the three Bayesian structures. Recall that equality of means is a Hypothesis Test in which a test statistic is calculated, and the means between two data sets are considered equal if the value that test statistic is less than the Student's T distribution value for the given sample size, and desired level of confidence. In Figure 4.27, the line labeled "Students T Value" is the threshold for equality of means for each of the software quality variables modeled, and for a 90% confidence. The remaining lines show the test statistic values calculated for each of the three Bayesian model structures proposed.

In general, all three of the Bayesian models did well in modeling the set of software quality variables with a mean that was equivalent to the mean of actual values. The Refined model shows some variables that were not modeled within this threshold, and the Intuitive structure reveals one software quality attribute (Failure Avoidance) that it could not model correctly. Of particular note is the number of missing attributes modeled by the Refined structure. This reflects the discovery that causation could not be established between phase correctness and completeness when the Intuitive Model was refined to eliminate insignificant

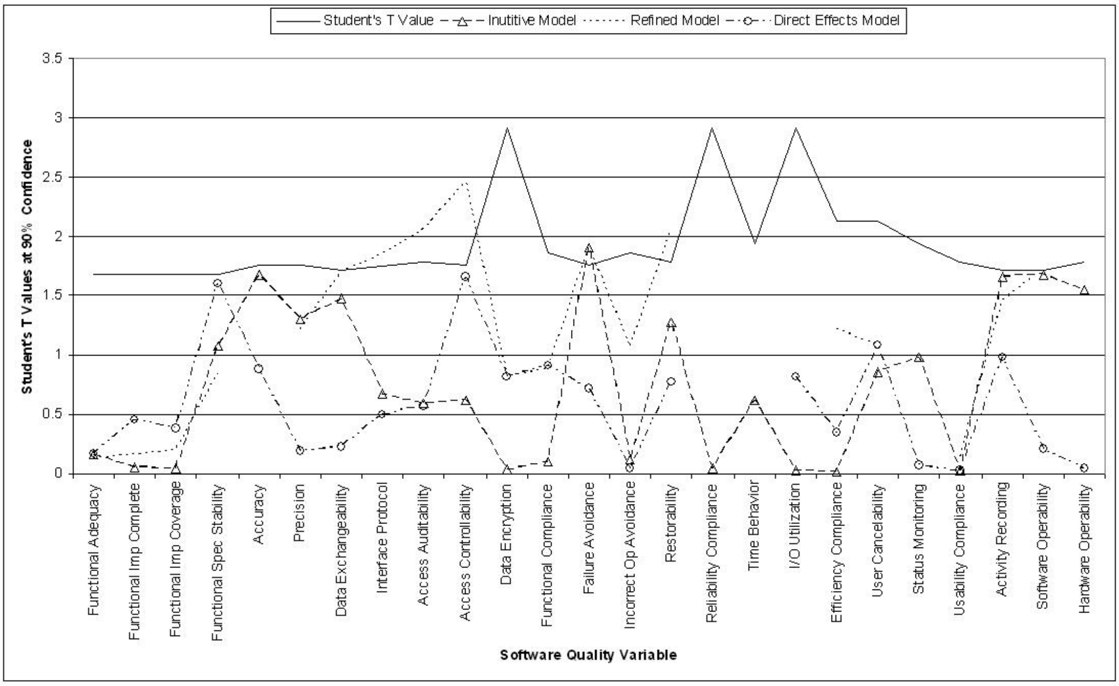

Figure 4.27: Equality of Means Results for Bayesian Model Structures

correlations. Of the three Bayesian approaches, the Direct Effects model structure passes the Equality of Means test for all variables it can model (two software quality variables are uncorrelated to the set of model inputs).

Figure 4.28 depicts the comparison of the three Bayesian models with respect to the Equality of Variances condition described in Table 3.19. As with the Equality of Means graph, the "Variance Threshold" line represents the threshold for equality of variances, and the remaining lines are the variance ratio values for each Bayesian model with respect to each software quality variable modeled. In cases of extreme variance ratio values, the maximum value was clamped to 15 in order to keep the chart readable. As with Equality of Means, the Direct Effects Model fits the actual data best in terms of variance. Only one

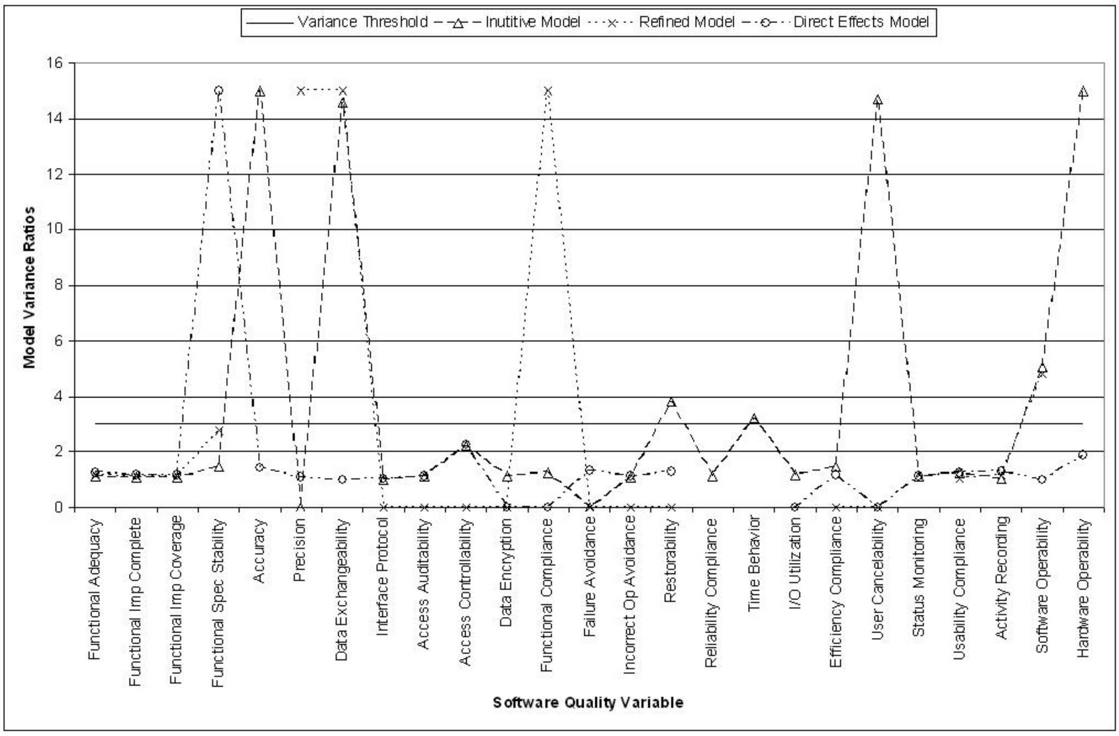

Figure 4.28: Equality of Variances Results for Bayesian Model Structures

software quality attribute (Functional Specification Stability) was modeled poorly by the Direct Effects Model.

Table 4.31 summarizes the comparison of accuracy of fit for the three model structures. Both the Intuitive and Direct Effects Models performed well with respect to fitting the set of software quality variables used in this research, although the Direct Effects Model was more accurate by comparison. The Refined Model, however, did not perform well. It was able to model the software quality data well for only 10 of the 25 variables.

Table 4.31: Summary of Accuracy of Fit Determinations for Bayesian Models

| ISO/IEC 9126 | Intuitive Model | | Refined Model | | Direct Effects Model | |
|---|---|---|---|---|---|---|
| Quality Metrics | Means | Vars | Means | Vars | Means | Vars |
| | Equal? | Equal? | Equal? | Equal? | Equal? | Equal? |
| **Functionality:** | | | | | | |
| Functional Adequacy | Yes | Yes | Yes | Yes | Yes | Yes |
| Functional Imp Complete | Yes | Yes | Yes | Yes | Yes | Yes |
| Functional Imp Coverage | Yes | Yes | Yes | Yes | Yes | Yes |
| Functional Spec Stablty | Yes | Yes | Yes | Yes | Yes | No |
| Accuracy | Yes | No | N/A | N/A | Yes | Yes |
| Precision | Yes | Yes | Yes | No | Yes | Yes |
| Data Exchangeability | Yes | No | Yes | No | Yes | Yes |
| Interface Protocol | Yes | Yes | No | Yes | Yes | Yes |
| Access Auditability | Yes | Yes | No | Yes | Yes | Yes |
| Access Controllability | Yes | Yes | No | Yes | Yes | Yes |
| Data Encryption | Yes | Yes | Yes | Yes | Yes | Yes |
| Functional Compliance | Yes | Yes | Yes | No | Yes | Yes |
| **Reliability:** | | | | | | |
| Failure Avoidance | No | Yes | No | Yes | Yes | Yes |
| Incorrect Op Avoidance | Yes | Yes | Yes | Yes | Yes | Yes |
| Restorability | Yes | No | No | Yes | Yes | Yes |
| Reliability Compliance | Yes | Yes | N/A | N/A | N/A | N/A |
| **Efficiency:** | | | | | | |
| Time Behavior | Yes | No | N/A | N/A | N/A | N/A |
| I/O Utilization | Yes | Yes | N/A | N/A | Yes | Yes |
| Efficiency Compliance | Yes | Yes | Yes | Yes | Yes | Yes |
| **Usability:** | | | | | | |
| User Op Cancelability | Yes | No | Yes | Yes | Yes | Yes |
| Op Status Monitoring | Yes | Yes | N/A | N/A | Yes | Yes |
| Usability Compliance | Yes | Yes | Yes | Yes | Yes | Yes |
| **Maintainability:** | | | | | | |
| Activity Recording | Yes | Yes | Yes | Yes | Yes | Yes |
| **Portability:** | | | | | | |
| Software Operability | Yes | No | No | No | Yes | Yes |
| Hardware Operability | Yes | No | N/A | N/A | Yes | Yes |

### *4.4.2 Predictive Validity Analysis*

Comparing the Intuitive, Refined and Direct Effects Model structures in terms of their Predictive Validity involves calculating the Average Relative Error (ARE) for each software quality variable as specified in Section 3.3.2.4. The ARE is a measure of model accuracy that describes the deviation of the predicted results from the actual results. The closer an ARE value is to zero, the more accurate the predictions. Figure 4.29 graphs the comparative ARE values for each of the three Bayesian model structures by modeled software quality attribute. From the diagram, it can be seen that the Intuitive Model is more accurate for 14 of the 25 software quality variables, and the Direct Effects model is more accurate for 10 of the software quality variables (for the Usability Compliance variables, both the Intuitive Model and the Direct Effects model have the same accuracy). The Refined Model structure is the least accurate of the three models.

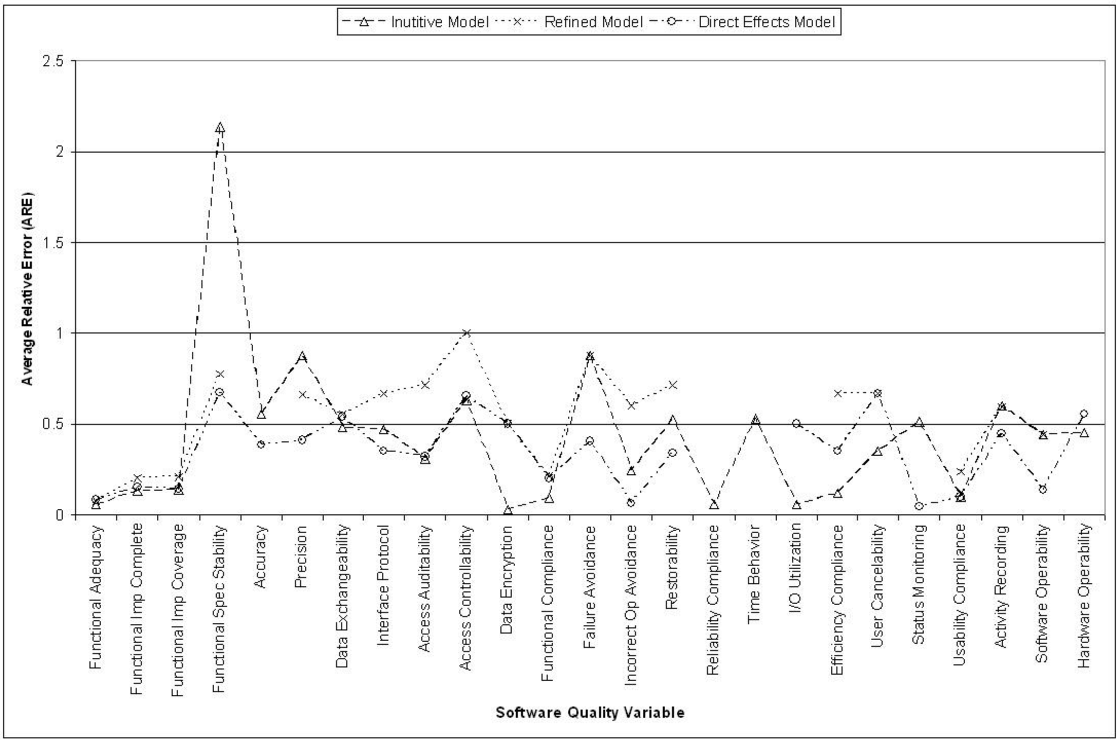

Figure 4.29: Average Relative Error Results for Bayesian Model Structures

Figure 4.30 shows a plot that compares the average belief, or confidence, associated with each model's software quality predictions. For 16 of the 25 software quality variables,

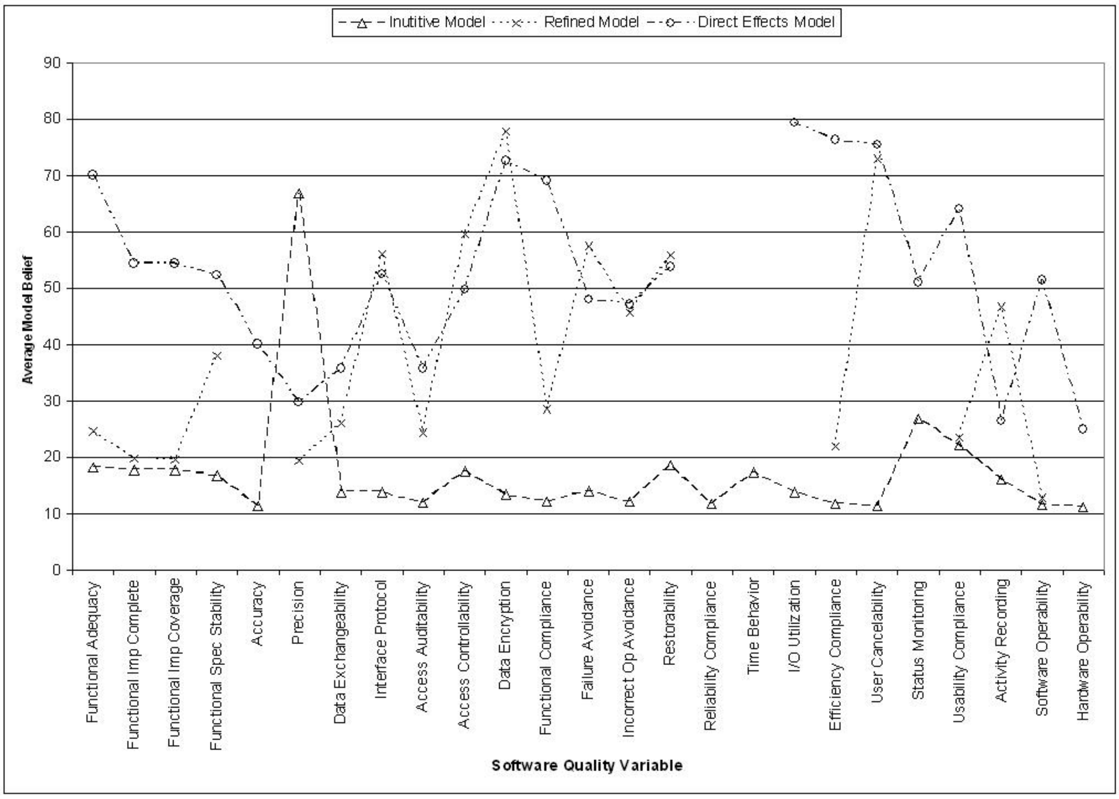

Figure 4.30: Average Belief Results for Bayesian Model Structures

the Direct Effects Model had the highest belief in its predictions. Belief is a factor for consideration when choosing the most appropriate model for each software quality variable. It gives the model's prediction credibility. A model in which the predictions are not as accurate, but the confidence in those predictions is high is often more desirable than a model with more accurate predictions, but less confidence.

Table 4.32 summarizes the comparison of the Bayesian models in terms of Predictive Validity. For most variables, the Intuitive Model provided the most accurate forecasts, but with low belief values. The Direct Effects model provided forecasts on the order of the

Intuitive Model, but had much higher belief values. The Refined Model performed poorly in predicting software quality.

Table 4.32: Summary of Predictive Validity Determinations for Bayesian Models

| ISO/IEC 9126 | Intuitive Model | | Refined Model | | Direct Effects Model | |
|---|---|---|---|---|---|---|
| Quality Metrics | ARE | Belief | ARE | Belief | ARE | Belief |
| **Functionality:** | | | | | | |
| Functional Adequacy | 0.062 | 18.28 | 0.076 | 24.69 | 0.085 | 70.04 |
| Functional Imp Complete | 0.l30 | 17.87 | 0.202 | 19.77 | 0.158 | 54.39 |
| Functional Imp Coverage | 0.139 | 17.86 | 0.208 | 19.73 | 0.143 | 54.39 |
| Functional Spec Stablty | 2.138 | 16.75 | 0.775 | 38.03 | 0.677 | 52.36 |
| Accuracy | 0.553 | 11.46 | N/A | N/A | 0.390 | 40.23 |
| Precision | 0.875 | 66.90 | 0.663 | 19.49 | 0.413 | 29.83 |
| Data Exchangeability | 0.481 | 13.81 | 0.556 | 26.01 | 0.544 | 35.74 |
| Interface Protocol | 0.471 | 13.96 | 0.667 | 56.10 | 0.353 | 52.46 |
| Access Auditability | 0.311 | 11.90 | 0.714 | 24.43 | 0.320 | 35.80 |
| Access Controllability | 0.633 | 17.60 | 1.000 | 59.59 | 0.659 | 49.78 |
| Data Encryption | 0.030 | 13.51 | 0.500 | 77.85 | 0.500 | 72.62 |
| Functional Compliance | 0.096 | 12.21 | 0.212 | 28.55 | 0.200 | 69.20 |
| **Reliability:** | | | | | | |
| Failure Avoidance | 0.875 | 14.06 | 0.875 | 57.57 | 0.405 | 47.98 |
| Incorrect Op Avoidance | 0.242 | 12.23 | 0.600 | 45.79 | 0.066 | 47.21 |
| Restorability | 0.578 | 18.64 | 0.714 | 55.91 | 0.341 | 53.89 |
| Reliability Compliance | 0.060 | 11.82 | N/A | N/A | N/A | N/A |
| **Efficiency:** | | | | | | |
| Time Behavior | 0.530 | 17.38 | N/A | N/A | N/A | N/A |
| I/O Utilization | 0.060 | 13.88 | N/A | N/A | 0.500 | 79.38 |
| Efficiency Compliance | 0.120 | 11.88 | 0.667 | 21.91 | 0.353 | 76.36 |
| **Usability:** | | | | | | |
| User Op Cancelability | 0.353 | 11.41 | 0.667 | 73.03 | 0.667 | 75.60 |
| Op Status Monitoring | 0.515 | 26.91 | N/A | N/A | 0.045 | 51.06 |
| Usability Compliance | 0.103 | 22.10 | 0.237 | 23.53 | 0.103 | 64.05 |
| **Maintainability:** | | | | | | |
| Activity Recording | 0.603 | 16.11 | 0.598 | 46.80 | 0.449 | 26.57 |
| **Portability:** | | | | | | |
| Software Operability | 0.442 | 11.51 | 0.444 | 12.80 | 0.140 | 51.44 |
| Hardware Operability | 0.454 | 11.27 | N/A | N/A | 0.554 | 25.02 |

### *4.4.3 Summary of Comparison of Bayesian Model Structures*

This section summarizes the results of the comparative analyses of the three proposed Bayesian software quality model structures. The Intuitive Model is a three-tiered model structure that first relates model inputs to intermediary nodes representing software life cycle phase correctness and completeness, and then relates those nodes to the various measures of software quality. The Refined Model streamlines the Intuitive Model to only include the cause-effect relationships that are verified through a correlative analysis. The Direct Effects Model structure is two-tiered, and simply relates inputs to software quality outputs.

Table 4.33 contains the model structures selected for each software quality variable. The selection of a model structure for a given software quality variable is based on the analyses performed in Sections 4.4.1 and 4.4.2. The model structure must have provided an accurate fit for modeled variables versus their actual values in terms of equality of means and variances. Also, the model structure had to produce software quality predictions that were relatively accurate, and carried with them a comparably high degree of confidence.

The Direct Effects Model structure was the most prominently selected. It offered the best combination of predictive accuracy and belief in that prediction. The Intuitive Model also was the selected structure option for several of the software quality variables. The Intuitive Model seems well-suited to model variables that are not correlated to any inputs, or model variables that have particularly small sample sizes. In practice, the Intuitive Model would likely be applied in just these types of circumstances, where little information is

Table 4.33: Selection of Bayesian Models for Software Quality Variables

| ISO/IEC 9126<br>Quality Metrics | Selected<br>Model | Justification |
|---|---|---|
| **Functionality:** | | |
| Functional Adequacy | Direct Effects | Acceptable accuracy, Highest belief |
| Functional Imp Complete | Direct Effects | Acceptable accuracy, Highest belief |
| Functional Imp Coverage | Direct Effects | Acceptable accuracy, Highest belief |
| Functional Spec Stablty | Refined | Best accuracy for Models that Fit |
| Accuracy | Direct Effects | Only Bayesian structure that Fit |
| Precision | Direct Effects | Highest accuracy, Acceptable belief |
| Data Exchangeability | Direct Effects | Only Bayesian structure that Fit |
| Interface Protocol | Direct Effects | Highest accuracy, Acceptable belief |
| Access Auditability | Direct Effects | Acceptable accuracy, Highest belief |
| Access Controllability | Direct Effects | Acceptable accuracy, Highest belief |
| Data Encryption | Intuitive | Highest accuracy |
| Functional Compliance | Intuitive | Highest accuracy |
| **Reliability:** | | |
| Failure Avoidance | Direct Effects | Only Bayesian structure that Fit |
| Incorrect Op Avoidance | Direct Effects | Highest accuracy, Highest belief |
| Restorability | Direct Effects | Only Bayesian structure that Fit |
| Reliability Compliance | Intuitive | Only Bayesian structure that Fit |
| **Efficiency:** | | |
| Time Behavior | Intuitive | Only Bayesian structure that Fit |
| I/O Utilization | Intuitive | Highest accuracy |
| Efficiency Compliance | Intuitive | Highest accuracy |
| **Usability:** | | |
| User Op Cancelability | Intuitive | Highest accuracy |
| Op Status Monitoring | Direct Effects | Highest accuracy, Highest belief |
| Usability Compliance | Direct Effects | Highest accuracy, Highest belief |
| **Maintainability:** | | |
| Activity Recording | Direct Effects | Highest accuracy, Acceptable belief |
| **Portability:** | | |
| Software Operability | Direct Effects | Only Bayesian structure that Fit |
| Hardware Operability | Direct Effects | Only Bayesian structure that Fit |

known about cause-effect relationships that affect a given software quality variable. In these situations, a model structure that incorporates all possible inputs is sensible to apply. It is expected that as more software engineering data is collected, and cause-effect relationships are more concretely identified, that the Direct Effects approach would emerge as a superior representation.

The poor performance of the Refined Model structure should not be construed as evidence against Kan's criteria for cause-effect relationships. After all, that same correlation criteria proved successful in identifying cause-effect relationships for the Direct Effects Model. As was suggested in the model variable analysis in Section 4.1.1, the phase correctness and completeness measures collected for this research were so invariant that they did not serve well as discriminators in a Bayesian Model. Once the Refined Model structure was determined through elimination of uncorrelated associations, the discriminating power that those phase correctness and completeness variables had as a set was whittled down to a single variable or two. In those cases, there simply was not enough variance to provide enough discriminating influence on the downstream software quality variables.

## 4.5 Comparison of Bayesian and Competing Models

In Section 4.4, a comparative analysis was performed between alternative Bayesian model structures in order to determine the structure that best modeled a set of software quality attributes. This same process is repeated in order to compare the selected Bayesian model structures with competing methods for modeling software quality. As described in Section 3.3.3, this research compares the Bayesian Belief Network with both the Least Squares Regression model and the Neural Network model. There is precedence for each of these methods as models of software quality (see Sections 2.2.2.2 and 2.2.3.2).

Least Squares Regression determines a set of weights for each independent variable, based on minimizing the Sum of Squared Error, in order to predict a dependent variable. Least Squares Regression represents a previous step in the evolution of software quality modeling. The intent of comparing Bayesian Belief Networks to Least Squares Regression is to demonstrate that BBNs are an improvement over Least Squares Regression in the modeling of software quality.

Neural Networks are similar to BBNs in that they can be represented as graphs of nodes that represent model inputs and outputs. Like BBNs, Neural Networks are adaptive systems that are trained, and then model based on prior data sets. In Neural Networks, training produces a weighting scheme for each input node to each output node. The intent of comparing Bayesian Belief Networks to Neural Networks is to demonstrate that BBNs are an improvement over other current methods of modeling software quality.

The approach to comparing the modeling techniques is similar to the approach used to compare the Bayesian models. Competing models will be compared in terms of their Accuracy of Fit to the data set, and in terms of their Predictive Validity. Accuracy of Fit will be measured through an analysis of the Equality of Means and Equality of Variances tests described in Section 3.3.2.3. The Accuracy of Fit tests identify those models that produce a mean and variance for modeled variables that is statistically equivalent to the actual mean and variance of those variables. The Predictive Validity test (see Section 3.3.2.4) is the measure of Average Relative Error which quantifies the deviation of the predicted values from the actual values of software quality. Because, like the Direct Effects Bayesian model, both the Least Squares Regression Model and the Neural Networks use correlated input variables, the Reliability Compliance and Time Behavior software quality variables will be ignored in this part of the analysis, as they had no correlated inputs.

### *4.5.1 Accuracy of Fit Analysis*

Determining the Accuracy of Fit for the three competing models involves identifying whether each technique models software quality variables such that the modeled values are equivalent to actual values. The technique for determining this is to verify that the mean and variance of the modeled values are statistically comparable to the mean and variance of the actual values, as described in Section 3.3.2.3. Figure 4.31 compares the Equality of Means results for each of the three modeling methods evaluated. The Equality of Means test involves the calculation of a test statistic which is expected to be less than the Student's T distribution value for the given sample size, and desired level of confidence if the means between two data sets are to be considered equal. In Figure 4.31, the line labeled ”Students T Value” is the threshold for equality of means for each of the software quality variables modeled, and for a 90% confidence. The remaining lines show the test statistic values calculated when each of the the three software quality modeling methods were used. For the Equality of Means test to be passed, the test statistic values should be less than the threshold line.

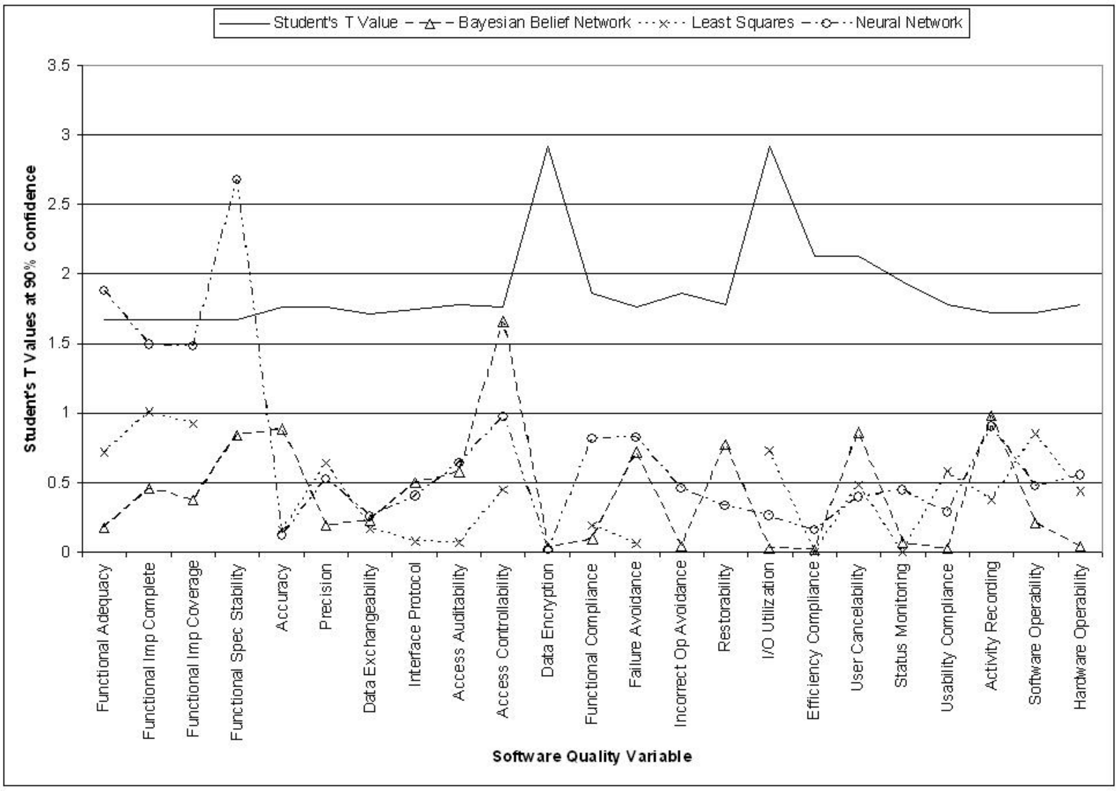

Figure 4.31: Equality of Means Results for Comparison of Software Quality Modeling Methods

Each of the three competing methods did well in modeling the set of software quality variables with a mean that was equivalent to the mean of actual values. The Neural Network shows some variables that were not modeled within this threshold. However, all three software quality modeling methods performed well. Of the three, the BBN is able to pass the Equality of Means test for all of the software quality variables. The Least Squares Regression model could not be applied to all software quality variables due to the presence of non-singular matrices, which makes a Least Squares prediction incalculable.

Figure 4.32 shows the comparison of the three competing software quality modeling methods with respect to the Equality of Variances condition described in Section 3.3.2.3. As

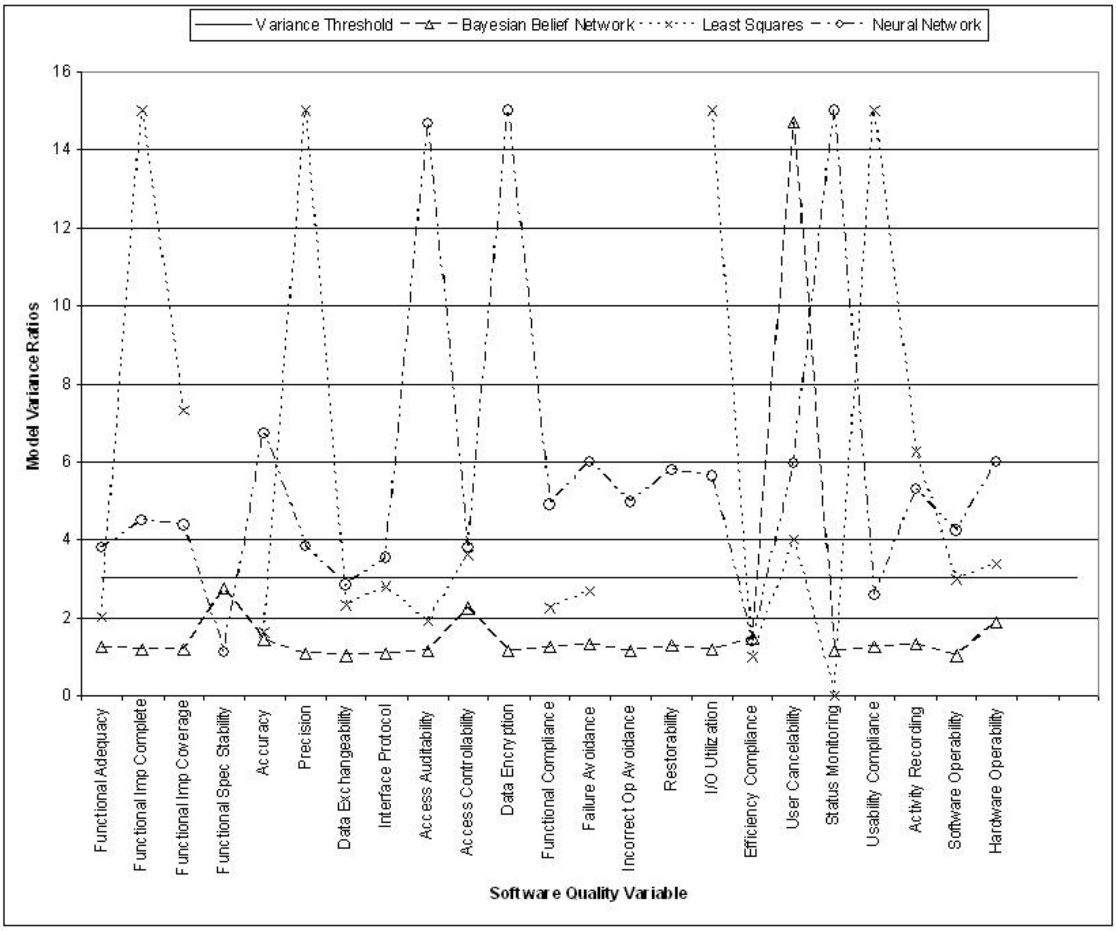

Figure 4.32: Equality of Variances Results for Comparison of Software Quality Modeling Methods

with the Equality of Means graph, the "Variance Threshold" line represents the threshold for equality of variances, and the remaining lines are the variance ratio values for each modeling technique with respect to each software quality variable. In cases of extreme variance ratio values, the maximum value was clamped to 15 in order to keep the chart readable.

The BBN model is clearly superior in terms of Equality of Variances. For all software quality variables except User Operation Cancellability the BBN satisfies the test for Equality of Variances. Least Squares Regression and Neural Networks by comparison modeled several variables as having a much different distribution than the actual data. Although the mean values of the Least Squares modeling method were accurate, the variances of the software quality variables were much less broad than it modeled. In the case of Neural Networks, the information presented in Figure 4.32 is misleading. While it is true that the ratio of variances for the Neural Network method, calculated as described in Table 3.19, produced a modeled distribution that did not vary with the actual distribution, the Neural Network model produced a variance that was more narrow than the actual distribution of software quality values. That is, the Neural Network modeled variables close to the mean value.

Table 4.34 summarizes the comparison of accuracy of fit for the three modeling approaches. While all three modeling techniques did well in modeling a distribution with a mean equivalent to the mean of the actual distribution, Bayesian Belief Networks outperformed both the Least Squares Regression and Neural Network models in terms of producing equivalent variances. The Neural Network did not produce variances that were equivalent, but in all cases produced a distribution with a much smaller variance than the actual data, indicating a tendency to model close to the mean.

### *4.5.2 Predictive Validity Analysis*

Comparing the BBN, Least Squares Regression and Neural Network models in terms of their Predictive Validity involves calculating the Average Relative Error (ARE) for each software quality variable which is a measure of model accuracy. Figure 4.33 graphs the

Table 4.34: Summary of Accuracy of Fit Determinations for Competing Models

| ISO/IEC 9126 | Bayesian Belief Net | | Least Squares Regression | | Neural Net | |
|---|---|---|---|---|---|---|
| Quality Metrics | Means | Vars | Means | Vars | Means | Vars |
| | Equal? | Equal? | Equal? | Equal? | Equal? | Equal? |
| **Functionality:** | | | | | | |
| Functional Adequacy | Yes | Yes | Yes | Yes | No | No |
| Functional Imp Complete | Yes | Yes | Yes | No | Yes | No |
| Functional Imp Coverage | Yes | Yes | Yes | No | Yes | No |
| Functional Spec Stablty | Yes | Yes | | | No | Yes |
| Accuracy | Yes | Yes | Yes | Yes | Yes | No |
| Precision | Yes | Yes | Yes | Yes | Yes | No |
| Data Exchangeability | Yes | Yes | Yes | Yes | Yes | No |
| Interface Protocol | Yes | Yes | Yes | Yes | Yes | No |
| Access Auditability | Yes | Yes | Yes | Yes | Yes | No |
| Access Controllability | Yes | Yes | Yes | No | Yes | No |
| Data Encryption | Yes | Yes | Yes | Yes | Yes | No |
| Functional Compliance | Yes | Yes | Yes | Yes | Yes | No |
| **Reliability:** | | | | | | |
| Failure Avoidance | Yes | Yes | Yes | Yes | Yes | No |
| Incorrect Op Avoidance | Yes | Yes | | | Yes | No |
| Restorability | Yes | Yes | | | Yes | No |
| **Efficiency:** | | | | | | |
| I/O Utilization | Yes | Yes | Yes | No | Yes | No |
| Efficiency Compliance | Yes | Yes | Yes | Yes | Yes | Yes |
| **Usability:** | | | | | | |
| User Op Cancelability | Yes | No | Yes | No | Yes | No |
| Op Status Monitoring | Yes | Yes | Yes | Yes | Yes | No |
| Usability Compliance | Yes | Yes | Yes | No | Yes | Yes |
| **Maintainability:** | | | | | | |
| Activity Recording | Yes | Yes | Yes | No | Yes | No |
| **Portability:** | | | | | | |
| Software Operability | Yes | Yes | Yes | Yes | Yes | No |
| Hardware Operability | Yes | Yes | Yes | No | Yes | No |

comparative ARE values for each of the three competing software quality modeling methods. As a lower value of ARE indicates higher accuracy in making predictions, it can be seen that the BBN model is more accurate in predicting software quality. That is, for most of the software quality variables model, the BBN provides the lowest ARE value.

Table 4.35 summarizes the comparison of the Bayesian, Least Squares, and Neural Network models in terms of Predictive Validity. For 22 of the 23 variables, the BBN provided

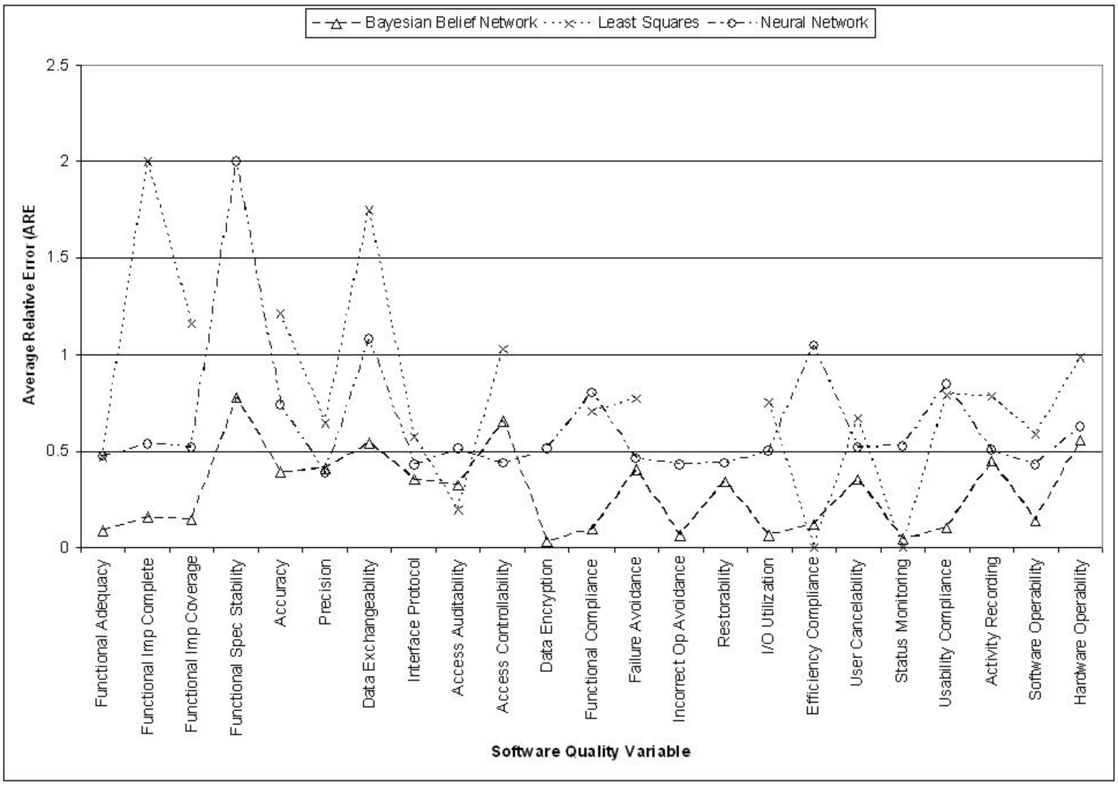

Figure 4.33: Average Relative Error Results for Comparison of Software Quality Modeling Methods

the most accurate forecasts. In addition, the 0.25 threshold for acceptable ARE proposed by Conte, et. al [CDS86], was met for 11 of the 23 modeled software quality variables.

### *4.5.3 Summary of Comparison of Bayesian and Competing Models*

This section summarizes the results of the comparative analysis of the three competing software quality modeling methods. The Bayesian Belief Network outperformed the Least Squares Regression Model and the Neural Network in terms of both Accuracy of Fit and Predictive Validity. That is, the Bayesian Belief Network modeled software quality data

Table 4.35: Summary of Predictive Validity Determinations for Competing Models

| ISO/IEC 9126<br>Quality Metrics | Bayesian<br>Belief Net | Least<br>Squares Regression | Neural<br>Net |
|---|---|---|---|
| **Functionality:** | | | |
| Functional Adequacy | 0.085 | 0.469 | 0.476 |
| Functional Imp Complete | 0.158 | 2.898 | 0.534 |
| Functional Imp Coverage | 0.143 | 1.161 | 0.520 |
| Functional Spec Stablty | 0.775 | 1.00 | 5.526 |
| Accuracy | 0.390 | 1.211 | 0.737 |
| Precision | 0.413 | 0.643 | 0.385 |
| Data Exchangeability | 0.544 | 1.751 | 1.081 |
| Interface Protocol | 0.353 | 0.576 | 0.432 |
| Access Auditability | 0.320 | 0.193 | 0.510 |
| Access Controllability | 0.659 | 1.027 | 0.437 |
| Data Encryption | 0.030 | N/A | 0.509 |
| Functional Compliance | 0.096 | 0.708 | 0.800 |
| **Reliability:** | | | |
| Failure Avoidance | 0.405 | 0.770 | 0.458 |
| Incorrect Op Avoidance | 0.066 | N/A | 0.429 |
| Restorability | 0.341 | N/A | 0.433 |
| **Efficiency:** | | | |
| I/O Utilization | 0.060 | 0.750 | 0.502 |
| Efficiency Compliance | 0.120 | N/A | 1.051 |
| **Usability:** | | | |
| User Op Cancelability | 0.353 | 0.667 | 0.515 |
| Op Status Monitoring | 0.045 | N/A | 0.526 |
| Usability Compliance | 0.103 | 0.792 | 0.845 |
| **Maintainability:** | | | |
| Activity Recording | 0.449 | 0.786 | 0.507 |
| **Portability:** | | | |
| Software Operability | 0.140 | 0.589 | 0.431 |
| Hardware Operability | 0.554 | 0.983 | 0.624 |

within acceptable thresholds of the actual data, and provided the most accurate predictions for unknown data sets. This result demonstrates that Bayesian Belief Networks, as a technique for modeling software quality, are an improvement over a prior popular approach to software quality modeling (Least Squares Regression), and perform better than at least one current approach to software quality modeling (Neural Networks).

Both the Least Squares Regression model and the Neural Network were weak in their ability to model the variances of the software quality variables. In the case of Least Squares Regression, the modeled variance was too broad, resulting in software quality forecasts that were accurate only when the set of actual software quality values happened to be close to their mean values. The Least Squares model was too reactive to value changes in the model inputs, and often produced forecasts that were well outside the actual value in terms of ARE. Interestingly, the Neural Network represents the other extreme in that it modeled the variances of software quality variables too narrowly. That is, the Neural Network was not sensitive enough to the value changes of the inputs, and produced predicted values that were too close to the mean value. The result was a set of ARE values that typically gravitated around 0.5.

## 4.6 Limitations of the Results

It is important to identify and discuss any elements of the sample environment or the analysis procedure which limit the applicability of the results. This section addresses those limitations.

### *4.6.1 Sample Size*

The validation of this model includes statistical techniques that accounted for the sample sizes of the various software quality variables. The Equality of Means Hypothesis Test, for example, uses the Student's T distribution in its calculations which accounts for the sample size being used to make the determination of equivalence. Despite this, however, it is recognized that the small sample sizes of several of these metrics are cause for concern. In the case of the Reliability Compliance metrics, for example, only two of the projects in the software engineering data set identified this as a quality need that was to be filled. Thus, the validation results for software quality variables with small sample sizes should be treated with caution.

### *4.6.2 Project Characteristics*

As described in Section 3.3.1.2, the software projects selected for this research are all small-scale development efforts. Team size varied from one to four software engineers, the number of source code files did not exceed 50 files, and the life cycle was typically completed inside a 3-4 month time period. While the results provide evidence that Bayesian Belief Networks are effective in modeling various attributes of software product quality, it is premature to infer that the results are applicable outside of the scope of the underlying software engineering data. A logical extension to this research is to apply this methodology to larger scale software development projects.

### *4.6.3 Effects of a Learning Environment*

This research acquired software engineering data from a combination of projects, including industry projects and those in an academic setting. While the student teams were comprised of both students and software engineering professionals, the intent of the class was to instruct students on the practices of software engineering. Thus, the artifacts for many projects were scrutinized in terms of the correctness of practically applying software engineering techniques, and not in terms of cost, schedule, or other industry drivers of engineering artifacts. It is possible that the ability of the model to provide accurate forecasts was affected by this learning environment. For example, all of the quality models had difficulty predicting for the Functional Specification Stability software quality attribute. In the process of developing a specification for a student project, that artifact, in many cases, incurred more changes than it would have in an industrial setting as the students refined their abilities to write succinct requirements. The additional changes associated with the student projects had the potential to skew the Functional Specification Stability measures for the set of projects used in this research.

# CHAPTER 5

# CONCLUSIONS

This research has proposed a model for software quality based on the cause-effect relationships that exist throughout the software development life cycle, and using Bayesian Belief Networks as the mechanism for relating software quality causal factors to variables that represent aspects of software quality. The purpose for developing the model is to address the need for a consistent approach to assessing and predicting software quality within a development project. The use of widely accepted and/or existing standards has been leveraged in order to provide a framework for quantifying the various drivers and indicators of software quality. The intent of this research is to establish a baseline for modeling quality in the software engineering life cycle, which may be further improved and extended with future research efforts. This section discusses the conclusions reached as a result of this research.

## 5.1 Causal Factor Frameworks

This research focused on the use of standards and industry in-use frameworks for quantifying causal factors that drive software quality, and for the variables that represent software quality. The subsections below provide discussion on the viability of the selected measurement frameworks in the context of software quality modeling. The major discriminators in these conclusions are the presence of multicollinearity among variables, and the framework to represent causal factors in a consistent and logical manner. The suitability of the various frameworks is focused largely on their ability to identify independent measurement categories that fit the acquired software engineering data.

### *5.1.1 Personnel Skill/Experience Framework*

This research used the industry in-use Competency Management System, fielded by NASA, as the basis for the assessment of skill and experience among software developers. This framework was found to model development team capability very effectively. Only two variables within the framework, Requirements Level 4 and Test Level 4, were found to model the same information. In addition to a lack of multicollinearity between skill variables in the same life cycle phase, there was only the one instance of multicollinearity between model variables across life cycle phases. This independence of variables gives the skill delineations credibility in terms of their ability to capture unique and discrete categories of skill.

The various correlation results were primarily consistent with expectations. In the identification of most significant causal factors of software quality, the presence of higher skill levels was correlated positively to software quality, and the presence of lower skill levels was correlated negatively to software quality. These results portray the expectation. It is the conclusion of this research that the proposed framework for measuring software personnel skill and experience is effective for modeling the capability of the development team in each phase of the software life cycle.

### *5.1.2 Process Maturity Framework*

The approach to quantifying process maturity for this research proved adequate, but not exceptional. In terms of multicollinearity, there was a large amount of overlap between variables. That is, several process variables were found to be modeling identical phenomena and so were ultimately consolidated. It is possible that this is a result of using small-scale projects as the basis for the data. Larger projects, in terms of size/complexity of the software and also team size, require more infrastructure to facilitate communication between groups of personnel. The ISO/IEC 15504, which was the standard used to categorize software engineering practices, is intended for the assessment of a wide range of project sizes. For the projects used in this research, the results indicate that such a broad set of practices is unnecessary.

### *5.1.3 Problem Complexity Framework*

This research proposed a framework for measuring problem complexity across several phases of the development life cycle. In the requirements phase, two complexity measures were introduced: quality need, which identified whether a the need existed in a project for a specific quality attribute, and quality coverage, which was an indicator of whether a given need was addressed in the specification. The design phase incorporated several different types of design measures that attempted to characterize the size of the designed software. The implementation phase focused on the proportion of developed source code that could be attributed to a given quality need as the complexity measure.

Of all of these measures, only those associated with the design phase or those representing changes to phase artifacts were significant. That is, only the design phase complexity variables, such as Depth of Inheritance Tree and Design Expansion, and the volatility measures for the requirements and design artifacts were found to be significant in the assessment and prediction of software quality. The measures of quality need and requirements coverage were quickly eliminated in the analysis of means and variances. The implementation phase complexity measure called prevalence, was significant only for one software quality variable.

Design measures and measures of change (volatility) are traditionally used in software quality prediction literature as discriminators for the influence of size and complexity on software quality. This research reinforces that position, and complements it with the knowledge that attempts at size and complexity measures in other software life cycle phases were found to be insignificant in terms of their correlation to software quality variables.

### *5.1.4 Software Product Quality Framework*

The ISO/IEC 9126 Software Product Quality standard was the basis for the measures that modeled software quality in this research. The ISO/IEC 9126 partitions software quality into logical categories, and provides specific measures for quantifying quality in each of those categories. For this study, 25 software quality variables were modeled using the measures described in the standard. The ISO/IEC 9126 standard was found to be effective in capturing the different aspects of software quality. The range of classification options were clearly defined such that assignment of a requirement to a quality category was a simple task. The classification of requirements in terms of the quality attributes provided an extra verification of the specification as those requirements that could be classified into multiple categories were deemed too complex and rewritten. Modeled quality variables were found to capture unique and independent aspects of software quality.

One impact in using the ISO/IEC 9126 standard is that it requires more fidelity in terms of categorizing requirements. Most of the quality measures proposed are simply proportions of the number of verified requirements in a quality category to the number of requirements in that categiory. Thus, it is helpful to organize the requirements specification according to the categories described in the standard. This allows the development team to classify both requirements, and the verification of requirements correctly. For a software organization with an institutionalized set of processes, this would simply involve an adjustment to the specification templates to accommodate the organization of requirements into the categories provided by the ISO/IEC 9126.

### *5.1.5 Availability of Software Engineering Data*

Both the construction and validation of the software quality model developed in this research relied heavily on the existence of software engineering data. Very early on in the planning for this study, it became apparent that any software engineering data needed would need to be specifically acquired. That is, the type of software engineering data needed to provide insight into this software quality model was not readily available. Of the few existing data sets that are freely available, none of them provided a level of detail necessary to give insight into the full complement of driving factors of quality. While most tracked various measures of size and complexity of the design problem or source code, there was no consistency in the measures selected, there was little measurement of factors outside of size and complexity, and there were no references to existing software quality standards.

Whether or not the measurement/modeling approaches presented in this study are widely adopted, it is clear from this research exercise that the software engineering community is in need of a standard measurement framework for quantifying the factors that influence software quality. It is the opinion of this researcher that the primary reason that software quality prediction models are not prevalently used is the lack of a standardized, validated, and applicable set of software engineering measures that quantifies the various driving factors of software quality.

## 5.2 Suitability of Bayesian Belief Networks

This research explored Bayesian Belief Networks as a mechanism for modeling software quality. BBNs model cause-effect relationships between variables, and quantify those relationships using conditional probabilities. Initially, three different approaches to BBN model structures were described and compared. The intent was to identify the optimum BBN structure for relating model inputs to outputs in the domain of software quality assessment and prediction. The resultant model was then compared to competing software quality model approaches. The criteria for comparison was an evaluation of the Accuracy of Fit of the modeled software quality data with respect to the actual software quality data, and an evaluation of the Predictive Validity of the the model's forecasts given a set of model inputs.

### *5.2.1 Comparison of Bayesian Model Structures*

Three different BBN model structures were proposed and compared in this research. The Intuitive Model structure was a three-tiered structure that related causal factors of software quality to software quality variables through an intermediary set of variables that represented quality of the artifacts of each software life cycle phase. This structure was based simply on a logical association of cause and effect. The Refined Model structure attempted to improve on the Intuitive Model structure by applying statistical rigor to each logical cause-effect relationship. The Direct Effects Model was a much simpler two-tiered model that related software quality variables to the set of model inputs that were significantly correlated.

Overall, the Direct Effects Model structure offered the best results in terms of all the criteria: Equality of Means and Variances, Average Relative Error, and in terms of the Belief, or level of confidence, in the results. This structure was found to produce the optimum results for 17 of the 25 software quality variables modeled. The Intuitive Model accounted for 7 of software quality attributes, and was found to produce the best results when sample sizes were low, or when no model inputs could be directly correlated to a model output. This is reasonable as the Intuitive structure provides a much more general model that is not biased by any correlated model inputs. That is, it is better suited to low sample size situations, where patterns have not yet emerged. In the results, a sample size of 5 projects was the boundary for transition between the Intuitive Model providing an optimum structure, and the Direct Effects Model providing an optimum structure. The Refined Model was found to produce the best results only for a single software quality variable, Functional Specification Stability. The expectations of this research were that the Refined Model would outperform the Intuitive Model as its internal structure was verified through an analysis of significant correlations. However, it was found that a lack of variance in the intermediary variables representing life cycle phase correctness and completeness masked the upstream influence of the model inputs.

### *5.2.2 Comparison of Bayesian and Competing Model Structures*

This research compared the Bayesian software quality modeling method to two other competing techniques: Least Squares Regression and Neural Networks. Least Squares Regression minimizes the sum of squared error values to produce a set of weights that are used to scale each input variable's influence on the modeled output. A Neural Network is an adaptive approach to modeling that weights an input's effect on the modeled output based on learning from a set of training data. For both Least Squares Regression and Neural Networks, the correlated causal factors associated with each software quality output were used as the model inputs (the same set of inputs used for the Direct Effects Bayesian model structure).

In terms of Accuracy of Fit, the three modeling techniques all performed well in terms of the Equality of Means Test. That is, the set of modeled software quality variables had an average value that was equivalent to the average value of the actual software quality measures. However, in terms of Equality of Variances, the Bayesian approach vastly outperformed the other methods. Least Squares Regression was found to be too reactive to model inputs, creating software quality variable distributions that were too broad. Conversely, the Neural Network was not sensitive enough to model inputs, and produced variances that were too narrow. Only the Bayesian models produced Accuracy of Fit results that passed both Equality of Means and Equality of Variances tests for 22 of the 23 variables used in the comparison.

This research has provided evidence that the use of Bayesian Belief Networks to model software quality is superior to both Least Squares Regression and Neural Networks. In addition to a better fit to the set of known data values, the Bayesian approach offered more accurate predictions for 22 of the 23 software quality variables used in the comparison. In addition, the Bayesian Belief Network model produced an Average Relative Error value of less than 0.25 for 11 of the 23 software quality variables. This is the threshold for acceptable ARE proposed by Conte, et. al [CDS86]. By comparison, modeling software quality using Least Squares Regression or Neural Networks could not achieve this threshold for any of the modeled variables.

### *5.2.3 System Resource Limitations of Bayesian Belief Networks*

The use of the Bayesian Belief Network as a modeling tool proved largely successful in terms of its ability to make accurate forecasts. However, there were several limitations that emerged during the process of implementing the model in the software. These limitations deal primarily with the consumption of system resources. A large amount of memory is required to implement a network of any significant size. The excessive memory consumption is inevitable as the Bayesian Belief Network increases in complexity. The state space is consumed quickly, particularly in cases where a given node in the BBN has several parents. The conditional probability table required to support such an arrangement requires consideration of all possible combinations of all possible states, and thus must calculate and store the probabilities associated with all combinations in memory. The BBNs constructed for this study routinely maximized the memory capacity of the machine on which they were executed (which had 1 Gigabyte of system memory). Although the simplification of the model structure discussed in Chapter 4 improved the model's consumption of the system memory, it remained an issue. The granularity of the Percentage Node discussed in Section 3.2.5.1 impacts both the precision of the model and the consumption of system memory. For this research, the balance between these two competing factors was realized with a maximum resolution of eight states. A host computer system with a larger memory capacity will allow for more discrete states in the output variables and thus improve the precision of the model's predictions.

Despite the complexity limitation due to memory capacity of the host machine, this study demonstrates that a BBN can be constructed that accurately models a complex software engineering problem. The encouraging aspect of this is that as technology improves, and higher performance computing machines are developed, complex BBNs can become an increasingly viable option for modeling. Machines with 64-bit memory addressing and 2 GHz processing speeds, available in today's market, could easily accommodate the BBNs developed for this research. While the near-term application of complex BBNs to represent software quality and other modeling problems may be sluggish in performance, there is enormous future potential for use of these models as the technology improves.

## 5.3 Primary Causes of Software Product Quality

Current trends in software engineering focus on process as the driving factor in software quality. The popular idea is that the institution of best practices in a development organization will lead to higher quality, better estimated software products (see Section 2.3.1 for a discussion of supporting studies). Government agencies, such as the National Aeronautics and Space Administration (NASA), require contracting organizations to be appraised at a certain process maturity level before considering their proposals [NAS04]. The perception is that a mature process brings with it an assurance of both process and product quality.

The results of this research do not address process quality, but do provide an alternate set of findings regarding product quality. The analysis performed in Section 4.3.7 clearly identifies personnel skill and experience as the principal set of factors in both software quality, and lack of software quality. That is, the capability of the development team had the most significant positive effect on the quality of the software product, and the most significant negative effect. Highly skilled personnel were correlated to high product quality, and less skilled personnel were correlated with poorer software quality. These results are vastly different than current convention which touts the development process as the primary driver for sound software quality.

The results indicate that the presence of less skilled developers has a negative influence on the product quality in a software development effort. This must be explored further, however, as it is infeasible to expect a project to completely comprised of expert personnel. The issue is not with the personnel, but with the assignment of tasks to those personnel. A developer with minimal competence in software development is typically expected to perform an equal share of the software construction responsibilities as his/her more highly skilled colleagues. This research concludes that those developers that have a minimal software skill set are assigned software development tasks that are inappropriately matched to their skill/experience level. This issue can be corrected through either an allocation of responsibilities appropriate to each team member's skill level, or possibly through a mentoring program that encourages developers to mature under to the tutelage of a more skilled and experienced team member.

If personnel skill and experience is most strongly correlated with software product quality, as this research asserts, then the effect of software process maturity warrants discussion. There is little doubt that software process improves the process quality of a development organization - this is the establishment of infrastructure that allows for more insight and understanding in the development process. The goal of process quality is to improve the management of the software development process and produce meaningful cost and schedule estimates. It is important to separate this type of quality, process quality, from quality in the software product itself. Software product quality addresses the quality in the delivered software system. How aspects of the process infrastructure affect the product quality were the focus of this research.

In Section 4.3.7, elements of the software process were found to significantly influence the various software product quality attributes, but with some inconsistencies. Verification practices, for example, influenced software quality both positively and negatively. On the surface, these results gives no clear direction on how to incorporate process into a software development effort in an effective way. This research asserts that the confused results are the result of process quality being the focus of the development effort instead of product quality. That is, project teams are interested in establishing process for the sake of the benefits gained from process quality and making assumptions that those practices will also benefit product quality. Intuitively, there is value in all of the practices modeled in this research as they relate to product quality. After all, it is difficult to argue against the practice of verifying artifacts. However, the issue seems one of application of the process, not of value of the process. That is, are the processes feasible to perform by the development team given the schedule and budget? This question seems to be the fulcrum of the issue. If the set of processes are planned appropriately, and the impact of those processes incorporated into the project schedule, then development teams can perform the process and still focus on the product, yielding a higher quality product. If processes are poorly planned or not complementary to the development team's product focus, then they become a distraction, and are a detriment to the product's quality.

One of the more interesting and unexpected results of analyzing the effects of process variables on variables of software quality was the positive effect that the Update Requirements variable had on software quality. Update Requirements is the practice of revisiting the requirements with the customer at the start of each iteration of an incremental life cycle. This result seems to be an indirect endorsement of an iterative approach to software development. Although for some processes, there was confusion in the results on whether certain practices positively or negatively affected software quality, it was clear that the repetitive review of the requirements was a positive influence.

The conclusion of this research in terms of factors that most significantly affect software quality is that projects should concentrate on acquiring the strongest set of skills for a development team in preference to attempting to engineer the best process or constrain the complexity of the software problem. The results indicate that skill and experience have the most significant effect on the software quality attributes measured in this study. Process maturity has the potential to positively influence software quality if the set of processes are complementary to the life cycle, and feasible to complete. This research affirms that software problem complexity affects software quality in a negative way, with changes in the requirements and design having the most dramatic adverse effect.

## 5.4 Model Applicability in the Software Life Cycle

There are multiple opportunities for application of this software quality model in the engineering life cycle. This model was developed primarily to assess and predict software quality. To that end, it would serve well in evolutionary life cycles such as the spiral model. Evolutionary life cycles iterate through development activities in order to mature the software product until it meets the customers' needs. The software quality model proposed in this research would functional well as a risk analysis tool at both the conclusion of the design phase, and at the conclusion of the test phase. Modeling at the conclusion of the design phase would give an opportunity for the development team to model their software effort, and react to any quality predictions made by the model. At the conclusion of the testing phase, the model would provide an accurate assessment of the state of the software product's quality. That information would be useful in the subsequent life cycle iteration to correct any identified quality deficiencies.

At the start of a software engineering project, the developed software quality model would be useful as an advisory tool for establishing development teams and processes. One of the benefits of Bayesian Belief Networks is they are bidirectional in terms of how they process variable states in the context of the training data. At the outset of a project, a project manager could use the model to input software quality values that are targets for the project. The underlying Bayesian Belief Network will process those values as inputs, and produce as outputs the skill/experience and process combination, based on that development organization's software engineering data, that is most likely to meet the designated target.

# APPENDIX A

# SOFTWARE SKILL ASSESSMENT QUESTIONNAIRE

## Individual Software Skill Assessment Form

Software Engineering

Skill Assessment Questionnaire

**Instructions**

This form is intended to assess your current level of education and experience in the field of software engineering. Please complete the questions below as accurately as possible. The information gathered here will be used as data to determine the effect of your skill level on the quality of your delivered software.

**Personal Information**

Name:

Occupation:

Years of industry software experience:

Contact email:

## Requirements Development

### Formal Training:

Number of academic/professional courses you have taken that has included requirements development as a subset of the curriculum:

Number of academic/professional courses you have taken in which the focus of the class was requirements/specification development:

Number of student projects in which you interacted with a user/customer in order to determine and document user/customer needs:

Number of student projects in which you developed a software specification:

### Experience:

Number of industry projects in which you interacted with a user/customer in order to determine and document user/customer needs:

Number of industry projects in which you developed a software specification:

Number of industry projects in which you served as the lead user/customer interface:

Number of industry projects in which you developed a system-level specifications (such as an interface specification defining the relationship between subsystem components):

Number of industry projects in which you developed a highly formalized (e.g., safety-critical) software specification:

Number of formal specification reviews in which you have participated:

Number of times you have presented a software specification for review:

## Software Design

### Formal Training:

Number of academic/professional courses you have taken that has included software design as a subset of the curriculum:

Number of academic/professional courses you have taken in which the focus of the class was software design:

Number of academic/professional courses you have taken that focused on a specialized form of software design (e.g., Database Design, design patterns, real-time software design, etc.):

Number of student projects in which you developed a software design and associated documentation:

### Experience:

Number of industry projects in which you developed a software design and associated documentation:

Number of industry projects in which you developed a specialized form of software design and associated documentation (e.g., Database Design, design patterns, real-time software design, etc.):

Number of industry projects in which you served as the lead for designing a software component of a larger system:

Number of industry projects in which you have served as the software architect (lead software designer) for a system with multiple components:

Number of formal design reviews in which you have participated:

Number of times you have presented a software design for review:

## Software Implementation

### Formal Training:

Number of academic/professional courses you have taken that has included software implementation as a subset of the curriculum (e.g., coding was required as part of a class project):

Number of academic/professional courses you have taken in which the focus of the class was software implementation (e.g., a class devoted to a specific programming language):

Number of student projects in which you implemented software:

### Experience:

Number of industry projects in which you implemented software:

For each major programming language below, list the number of projects in which you used the language:

C/C++

Java

Visual Basic

List any other languages/packages/products/protocols/etc that you are familiar with (e.g., SQL, HTML, TCP/IP, VxWorks, Python):

Number of industry projects in which you served as the lead for implementing a software component in a larger system:

Number of formal code reviews in which you have participated:

Number of times you have presented source code for review:

## Software Integration and Test

### Formal Training:

Number of academic/professional courses you have taken that has included software testing as a subset of the curriculum:

Number of academic/professional courses you have taken in which the focus of the class was software testing:

Number of student projects in which you interacted with a user/customer in order to determine and document user/customer needs:

Number of student projects in which you developed software test plans and/or procedures:

Number of student projects in which you executed a software test procedure:

**Experience:**

Number of industry projects in which you developed software test plans and/or procedures::

Number of industry projects in which you executed a software test procedure:

Number of industry projects in which you served as the lead for testing a software component in a larger system:

Number of industry projects in which you served as the system test lead for a software system with multiple components:

Number of industry projects in which you served as a quality assurance specialist for the testing of a software package:

# APPENDIX B

# DIRECT EFFECTS ANALYSIS TABLES

The following tables capture the analysis of the direct effects of each of the various potential causal factors of software quality on the software quality variables represented in the developed model. The tables are organized by category (Development Team Skill, Process Maturity, and Problem Complexity) and then by life cycle phase (Requirements, Design, Implementation, Integration/Test). These tables form the basis for the analysis performed in Section 4.3.

Table B.1: Correlation of Requirements Skill Level 1 to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Req Skill 1<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | -0.1027 | 0.7447 | 1.6726 | No |
| Functional Completeness | 28 | -0.0966 | 0.7130 | 1.6726 | No |
| Functional Coverage | 28 | -0.1329 | 0.9851 | 1.6726 | No |
| Specification Stability | 28 | -0.1066 | 0.7878 | 1.6726 | No |
| Accuracy | 9 | 0.2988 | 1.2522 | 1.7340 | No |
| Precision | 9 | 0.3705 | 1.5955 | 1.7340 | No |
| Data Exchangeability | 13 | -0.0860 | 0.4227 | 1.7060 | No |
| Interface Consistency | 9 | 0.0062 | 0.0249 | 1.7340 | No |
| Access Auditability | 7 | 0.0430 | 0.1492 | 1.7610 | No |
| Functional Compliance | 5 | -0.2500 | 0.7303 | 1.8120 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.3031 | 1.1901 | 1.7460 | No |
| **Usability** | | | | | |
| User Cancelability | 3 | 0.3333 | 0.8660 | 1.8600 | No |
| Status Monitoring | 4 | -0.2500 | 0.7303 | 1.8120 | No |
| Usability Compliance | 7 | -0.6455 | 2.9277 | 1.7610 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | 0.2524 | 1.2780 | 1.7060 | No |

Table B.2: Correlation of Requirements Skill Level 2 to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Req Skill 2<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | 0.0151 | 0.1092 | 1.6726 | No |
| Functional Completeness | 28 | 0.0267 | 0.1961 | 1.6726 | No |
| Functional Coverage | 28 | 0.0333 | 0.2447 | 1.6726 | No |
| Specification Stability | 28 | 0.3142 | 2.4319 | 1.6726 | Yes |
| Accuracy | 9 | -0.2011 | 0.8213 | 1.7340 | No |
| Precision | 9 | -0.0950 | 0.3815 | 1.7340 | No |
| Data Exchangeability | 13 | 0.5894 | 3.5743 | 1.7060 | Yes |
| Interface Consistency | 9 | -0.2798 | 1.1658 | 1.7340 | No |
| Access Auditability | 7 | -0.1698 | 0.5968 | 1.7610 | No |
| Access Controllability | 8 | 0.5229 | 2.2953 | 1.7460 | Yes |
| Functional Compliance | 5 | 0.5145 | 1.6971 | 1.8120 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | -0.4088 | 1.6760 | 1.8120 | Yes |
| Restorability | 7 | -0.4470 | 1.7312 | 1.7610 | No |
| **Usability** | | | | | |
| User Cancelability | 3 | -0.3333 | 0.8660 | 1.8600 | No |
| Status Monitoring | 4 | -0.7906 | 3.6515 | 1.8120 | Yes |
| Usability Compliance | 7 | -0.3063 | 1.1145 | 1.7610 | No |
| **Maintainability** | | | | | |
| Data Logging | 12 | 0.0117 | 0.0576 | 1.7060 | No |
| **Portability** | | | | | |
| Software Operability | 13 | 0.1882 | 0.9386 | 1.7060 | No |
| Hardware Operability | 7 | 0.3341 | 1.3263 | 1.7460 | No |

Table B.3: Correlation of Requirements Skill Level 3 to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Req Skill 3<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | 0.0250 | 0.1804 | 1.6726 | No |
| Functional Completeness | 28 | -0.0463 | 0.3408 | 1.6726 | No |
| Functional Coverage | 28 | -0.0349 | 0.2566 | 1.6726 | No |
| Specification Stability | 28 | -0.1932 | 1.4470 | 1.6726 | No |
| Accuracy | 9 | -0.0796 | 0.3194 | 1.7340 | No |
| Precision | 9 | -0.1448 | 0.5855 | 1.7340 | No |
| Data Exchangeability | 13 | -0.5609 | 3.3188 | 1.7060 | Yes |
| Interface Consistency | 9 | 0.2095 | 0.8571 | 1.7340 | No |
| Access Auditability | 7 | 0.2582 | 0.9258 | 1.7610 | No |
| Access Controllability | 8 | 0.3780 | 1.5275 | 1.7460 | No |
| Functional Compliance | 5 | -0.4564 | 1.4510 | 1.8120 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.3031 | 1.1901 | 1.7460 | No |
| Incorrect Op Avoidance | 5 | -0.5590 | 1.9069 | 1.8120 | Yes |
| Restorability | 7 | 0.2810 | 1.0142 | 1.7610 | No |
| **Usability** | | | | | |
| Status Monitoring | 4 | 0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | 0.3000 | 1.0894 | 1.7610 | No |
| **Maintainability** | | | | | |
| Data Logging | 12 | -0.3465 | 1.8096 | 1.7060 | Yes |
| **Portability** | | | | | |
| Software Operability | 13 | -0.4799 | 2.6799 | 1.7060 | Yes |
| Hardware Operability | 7 | -0.5774 | 2.6458 | 1.7460 | Yes |

Table B.4: Correlation of Requirements Skill Level 4 to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Req Skill 4<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | 0.1799 | 1.3189 | 1.6726 | No |
| Functional Completeness | 28 | 0.3493 | 2.7390 | 1.6726 | Yes |
| Functional Coverage | 28 | 0.3995 | 3.2022 | 1.6726 | Yes |
| Specification Stability | 28 | -0.2369 | 1.7918 | 1.6726 | Yes |
| Precision | 9 | -0.8047 | 5.4210 | 1.7340 | Yes |
| Data Exchangeability | 13 | -0.3886 | 2.0660 | 1.7060 | Yes |
| Interface Consistency | 9 | 0.3001 | 1.2585 | 1.7340 | No |
| Access Controllability | 8 | 0.2928 | 1.1456 | 1.7460 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.2037 | 0.7783 | 1.7460 | No |
| Restorability | 7 | 0.7024 | 3.4188 | 1.7610 | Yes |
| **Usability** | | | | | |
| Status Monitoring | 4 | 0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | 0.7303 | 3.7033 | 1.7610 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | 0.2855 | 1.4595 | 1.7060 | No |

Table B.5: Correlation of Design Skill Level 1 to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Des Skill 1<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | -0.1120 | 0.8124 | 1.6726 | No |
| Functional Completeness | 28 | 0.1827 | 1.3656 | 1.6726 | No |
| Functional Coverage | 28 | 0.1403 | 1.0413 | 1.6726 | No |
| Specification Stability | 28 | 0.2582 | 1.9642 | 1.6726 | Yes |
| Accuracy | 9 | 0.1427 | 0.5768 | 1.7340 | No |
| Precision | 9 | 0.2765 | 1.1508 | 1.7340 | No |
| Data Exchangeability | 13 | 0.1429 | 0.7076 | 1.7060 | No |
| Interface Consistency | 9 | -0.6656 | 3.5678 | 1.7340 | Yes |
| Access Auditability | 7 | -0.1557 | 0.5460 | 1.7610 | No |
| Functional Compliance | 5 | -0.2500 | 0.7303 | 1.8120 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.3031 | 1.1901 | 1.7460 | No |
| Incorrect Op Avoidance | 5 | -0.5590 | 1.9069 | 1.8120 | Yes |
| Restorability | 7 | -0.5960 | 2.5714 | 1.7610 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | -0.4237 | 2.2916 | 1.7060 | Yes |
| **Portability** | | | | | |
| Software Operability | 13 | 0.4702 | 2.6103 | 1.7060 | Yes |
| Hardware Operability | 7 | 0.5625 | 2.5459 | 1.7460 | Yes |

Table B.6: Correlation of Design Skill Level 2 to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Des Skill 2<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | -0.1172 | 0.8509 | 1.6726 | No |
| Functional Completeness | 28 | -0.3548 | 2.7887 | 1.6726 | Yes |
| Functional Coverage | 28 | -0.3813 | 3.0308 | 1.6726 | Yes |
| Specification Stability | 28 | 0.0698 | 0.5140 | 1.6726 | No |
| Accuracy | 9 | -0.3087 | 1.2983 | 1.7340 | No |
| Precision | 9 | -0.0073 | 0.0293 | 1.7340 | No |
| Data Exchangeability | 13 | 0.2115 | 1.0602 | 1.7060 | No |
| Interface Consistency | 9 | 0.1894 | 0.7714 | 1.7340 | No |
| Access Auditability | 7 | 0.1557 | 0.5460 | 1.7610 | No |
| Access Controllability | 8 | -0.2928 | 1.1456 | 1.7460 | No |
| Data Encryption | 2 | 1.0000 | N/A | 2.1320 | Yes |
| Functional Compliance | 5 | 0.3953 | 1.2172 | 1.8120 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | -0.4088 | 1.6760 | 1.7460 | No |
| Incorrect Op Avoidance | 5 | -0.2795 | 0.8234 | 1.8120 | No |
| Restorability | 7 | -0.7476 | 3.8998 | 1.7610 | Yes |
| **Efficiency** | | | | | |
| Efficiency Compliance | 3 | 1.0000 | N/A | 1.9430 | Yes |
| **Usability** | | | | | |
| User Cancelability | 3 | 1.0000 | N/A | 1.8600 | Yes |
| Status Monitoring | 4 | -0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | -0.7303 | 3.7033 | 1.7610 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | -0.1551 | 0.7689 | 1.7060 | No |
| **Portability** | | | | | |
| Software Operability | 13 | -0.2328 | 1.1728 | 1.7060 | No |
| Hardware Operability | 7 | -0.2529 | 0.9779 | 1.7460 | No |

Table B.7: Correlation of Design Skill Level 3 to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Des Skill 3<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | 0.1672 | 1.2229 | 1.6726 | No |
| Functional Completeness | 28 | 0.1580 | 1.1761 | 1.6726 | No |
| Functional Coverage | 28 | 0.2077 | 1.5599 | 1.6726 | No |
| Specification Stability | 28 | -0.2184 | 1.6446 | 1.6726 | No |
| Accuracy | 9 | 0.3966 | 1.7281 | 1.7340 | No |
| Precision | 9 | -0.1448 | 0.5855 | 1.7340 | No |
| Data Exchangeability | 13 | -0.1982 | 0.9906 | 1.7060 | No |
| Interface Consistency | 9 | 0.3001 | 1.2585 | 1.7340 | No |
| Access Controllability | 8 | 0.2928 | 1.1456 | 1.7460 | No |
| Functional Compliance | 5 | -0.2500 | 0.7303 | 1.8120 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.2037 | 0.7783 | 1.7460 | No |
| Incorrect Op Avoidance | 5 | 0.5590 | 1.9069 | 1.8120 | Yes |
| Restorability | 7 | 0.8278 | 5.1120 | 1.7610 | Yes |
| **Usability** | | | | | |
| Status Monitoring | 4 | 0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | 0.7303 | 3.7033 | 1.7610 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | 0.3673 | 1.9348 | 1.7060 | Yes |
| **Portability** | | | | | |
| Software Operability | 13 | -0.1777 | 0.8845 | 1.7060 | No |
| Hardware Operability | 7 | 0.0000 | N/A | 1.7460 | No |

Table B.8: Correlation of Design Skill Level 4 to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Des Skill 4<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | 0.1975 | 1.4527 | 1.6726 | No |
| Functional Completeness | 28 | 0.3079 | 2.3783 | 1.6726 | Yes |
| Functional Coverage | 28 | 0.3627 | 2.8605 | 1.6726 | Yes |
| Specification Stability | 28 | -0.2608 | 1.9848 | 1.6726 | Yes |
| Accuracy | 9 | -0.3752 | 1.6188 | 1.7340 | No |
| Precision | 9 | -0.8047 | 5.4210 | 1.7340 | Yes |
| Data Exchangeability | 13 | -0.3886 | 2.0660 | 1.7060 | Yes |
| Interface Consistency | 9 | 0.3001 | 1.2585 | 1.7340 | No |
| Access Controllability | 8 | 0.2928 | 1.1456 | 1.7460 | No |
| Functional Compliance | 5 | -0.2500 | 0.7303 | 1.8120 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.3069 | 1.2067 | 1.7460 | No |
| Restorability | 7 | 0.7024 | 3.4188 | 1.7610 | Yes |
| **Usability** | | | | | |
| Status Monitoring | 4 | 0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | 0.7303 | 3.7033 | 1.7610 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | 0.2855 | 1.4595 | 1.7060 | No |
| **Portability** | | | | | |
| Software Operability | 13 | 0.2417 | 1.2201 | 1.7060 | No |

Table B.9: Correlation of Implementation Skill Level 1 to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Imp Skill 1<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | -0.0688 | 0.4975 | 1.6726 | No |
| Functional Completeness | 28 | 0.1513 | 1.1249 | 1.6726 | No |
| Functional Coverage | 28 | 0.1246 | 0.9227 | 1.6726 | No |
| Specification Stability | 28 | -0.0236 | 0.1733 | 1.6726 | No |
| Accuracy | 9 | 0.1427 | 0.5768 | 1.7340 | No |
| Precision | 9 | 0.4096 | 1.7962 | 1.7340 | Yes |
| Data Exchangeability | 13 | 0.3404 | 1.7738 | 1.7060 | Yes |
| Access Auditability | 7 | 0.2582 | 0.9258 | 1.7610 | No |
| Access Controllability | 8 | 0.1429 | 0.5401 | 1.7460 | No |
| Functional Compliance | 5 | -0.2500 | 0.7303 | 1.8120 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.2694 | 1.0467 | 1.7460 | No |
| **Usability** | | | | | |
| User Cancelability | 3 | 0.3333 | 0.8660 | 1.8600 | No |
| **Maintainability** | | | | | |
| Data Logging | 12 | -0.2408 | 1.2153 | 1.7060 | No |
| **Portability** | | | | | |
| Software Operability | 13 | 0.3136 | 1.6181 | 1.7060 | No |
| Hardware Operability | 7 | -0.3780 | 1.5275 | 1.7460 | No |

Table B.10: Correlation of Implementation Skill Level 2 to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Imp Skill 2<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | 0.0627 | 0.4531 | 1.6726 | No |
| Functional Completeness | 28 | -0.2979 | 2.2933 | 1.6726 | Yes |
| Functional Coverage | 28 | -0.2805 | 2.1477 | 1.6726 | Yes |
| Specification Stability | 28 | 0.3216 | 2.4962 | 1.6726 | Yes |
| Accuracy | 9 | -0.0544 | 0.2181 | 1.7340 | No |
| Precision | 9 | -0.0370 | 0.1481 | 1.7340 | No |
| Data Exchangeability | 13 | 0.2397 | 1.2097 | 1.7060 | No |
| Interface Consistency | 9 | -0.1380 | 0.0399 | 1.7340 | No |
| Access Auditability | 7 | 0.1698 | 0.5968 | 1.7610 | No |
| Access Controllability | 8 | -0.4587 | 1.9314 | 1.7460 | Yes |
| Data Encryption | 2 | 1.0000 | N/A | 2.1320 | Yes |
| Functional Compliance | 5 | 0.5833 | 2.0313 | 1.8120 | Yes |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | -0.4073 | 1.6687 | 1.7460 | No |
| Incorrect Op Avoidance | 5 | -0.8385 | 4.3529 | 1.8120 | Yes |
| Restorability | 7 | -0.8158 | 4.8863 | 1.7610 | No |
| **Efficiency** | | | | | |
| I/O Utilization | 2 | 1.0000 | N/A | 2.1320 | Yes |
| Efficiency Compliance | 3 | 1.0000 | N/A | 1.9430 | Yes |
| **Usability** | | | | | |
| User Cancelability | 3 | 0.5773 | 1.7321 | 1.8600 | No |
| Status Monitoring | 4 | -0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | -0.4528 | 1.7593 | 1.7610 | No |
| **Maintainability** | | | | | |
| Data Logging | 12 | 0.0693 | 0.3403 | 1.7060 | No |
| **Portability** | | | | | |
| Software Operability | 13 | 0.2495 | 1.2620 | 1.7060 | No |
| Hardware Operability | 7 | 0.4437 | 1.8524 | 1.7460 | Yes |

Table B.11: Correlation of Implementation Skill Level 3 to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Imp Skill 3<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Functionality** | | | | | |
| Functional Adequacy | 28 | -0.0326 | 0.2352 | 1.6726 | No |
| Functional Completeness | 28 | -0.0804 | 0.5926 | 1.6726 | No |
| Functional Coverage | 28 | 0.0846 | 0.6239 | 1.6726 | No |
| Specification Stability | 28 | -0.1375 | 1.0202 | 1.6726 | No |
| Accuracy | 9 | 0.0162 | 0.0648 | 1.7340 | No |
| Precision | 9 | 0.0475 | 0.1904 | 1.7340 | No |
| Data Exchangeability | 13 | -0.2596 | 1.3171 | 1.7060 | No |
| Interface Consistency | 9 | 0.0868 | 0.5571 | 1.7340 | No |
| Access Auditability | 7 | -0.1557 | 0.5460 | 1.7610 | No |
| Access Controllability | 8 | 0.3394 | 1.3502 | 1.7460 | No |
| Data Encryption | 2 | 1.0000 | N/A | 2.1320 | Yes |
| Functional Compliance | 5 | -0.3430 | 1.0328 | 1.8120 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.2037 | 0.7783 | 1.7460 | No |
| Incorrect Op Avoidance | 5 | 0.8385 | 4.3529 | 1.8120 | Yes |
| Restorability | 7 | 0.3083 | 1.1228 | 1.7610 | No |
| **Usability** | | | | | |
| Status Monitoring | 4 | 0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | -0.7156 | 3.5491 | 1.7610 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | -0.1626 | 0.8076 | 1.7060 | No |
| **Portability** | | | | | |
| Software Operability | 13 | -0.4696 | 2.6055 | 1.7060 | Yes |
| Hardware Operability | 7 | -0.1796 | 0.6831 | 1.7460 | No |

Table B.12: Correlation of Implementation Skill Level 4 to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Imp Skill 4<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Functionality** | | | | | |
| Functional Adequacy | 28 | 0.0179 | 0.1292 | 1.6726 | No |
| Functional Completeness | 28 | 0.3550 | 2.7904 | 1.6726 | Yes |
| Functional Coverage | 28 | 0.3627 | 2.8561 | 1.6726 | Yes |
| Specification Stability | 28 | -0.2771 | 2.1194 | 1.6726 | Yes |
| Accuracy | 9 | 0.3752 | 1.6188 | 1.7340 | No |
| Precision | 9 | -0.8047 | 5.4210 | 1.7340 | Yes |
| Data Exchangeability | 13 | -0.3886 | 2.0660 | 1.7060 | Yes |
| Interface Consistency | 9 | -0.4610 | 0.3487 | 1.7340 | No |
| Access Controllability | 8 | 0.2928 | 1.1456 | 1.7460 | No |
| Functional Compliance | 5 | -0.4082 | 1.2649 | 1.8120 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.2843 | 1.1096 | 1.7460 | No |
| Restorability | 7 | 0.7024 | 3.4188 | 1.7610 | Yes |
| **Usability** | | | | | |
| Status Monitoring | 4 | 0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | 0.7303 | 3.7033 | 1.7610 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | 0.2855 | 1.4595 | 1.7060 | No |
| **Portability** | | | | | |
| Software Operability | 13 | 0.1169 | 0.5768 | 1.7060 | No |

Table B.13: Correlation of Integration/Test Skill Level 1 to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Tst Skill 1<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Functionality** | | | | | |
| Functional Adequacy | 28 | 0.0301 | 0.2169 | 1.6726 | No |
| Functional Completeness | 28 | -0.0400 | 0.2941 | 1.6726 | No |
| Functional Coverage | 28 | 0.0416 | 0.3057 | 1.6726 | No |
| Specification Stability | 28 | -0.0372 | 0.2736 | 1.6726 | Yes |
| Accuracy | 9 | 0.0064 | 0.0256 | 1.7340 | No |
| Precision | 9 | 0.5068 | 2.3517 | 1.7340 | No |
| Data Exchangeability | 13 | 0.1450 | 0.7181 | 1.7060 | No |
| Interface Consistency | 9 | -0.2651 | 1.0999 | 1.7340 | No |
| Access Auditability | 7 | 0.5119 | 2.0641 | 1.7610 | Yes |
| Access Controllability | 8 | -0.6064 | 2.8538 | 1.7460 | Yes |
| Functional Compliance | 5 | -0.2500 | 0.7303 | 1.8120 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.4608 | 1.9426 | 1.7460 | Yes |
| Incorrect Op Avoidance | 5 | 0.0000 | N/A | 1.8120 | No |
| Restorability | 7 | -0.9234 | 8.3324 | 1.7610 | Yes |
| **Usability** | | | | | |
| User Cancelability | 3 | 0.5774 | 1.7321 | 1.8600 | No |
| Status Monitoring | 4 | -0.2500 | 0.7303 | 1.8120 | No |
| Usability Compliance | 7 | -0.6455 | 2.9277 | 1.7610 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | -0.3594 | 1.8865 | 1.7060 | Yes |
| **Portability** | | | | | |
| Software Operability | 13 | 0.0116 | 0.0568 | 1.7060 | No |
| Hardware Operability | 7 | -0.2097 | 0.8023 | 1.7460 | No |

Table B.14: Correlation of Integration/Test Skill Level 2 to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Tst Skill 2<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | -0.2838 | 2.1340 | 1.6726 | Yes |
| Functional Completeness | 28 | -0.1505 | 1.1188 | 1.6726 | No |
| Functional Coverage | 28 | -0.2247 | 1.6943 | 1.6726 | Yes |
| Specification Stability | 28 | 0.4083 | 3.2866 | 1.6726 | Yes |
| Accuracy | 9 | -0.5266 | 2.4781 | 1.7340 | Yes |
| Precision | 9 | 0.0719 | 0.2883 | 1.7340 | No |
| Data Exchangeability | 13 | 0.1094 | 0.5393 | 1.7060 | No |
| Interface Consistency | 9 | 0.1155 | 0.4651 | 1.7340 | No |
| Access Auditability | 7 | -0.5675 | 2.0641 | 1.7610 | Yes |
| Access Controllability | 8 | 0.2757 | 1.0730 | 1.7460 | No |
| Data Encryption | 2 | 1.0000 | N/A | 2.1320 | Yes |
| Functional Compliance | 5 | 0.3953 | 1.2172 | 1.8120 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.0272 | 0.1020 | 1.7460 | No |
| Incorrect Op Avoidance | 5 | 0.7500 | 3.2071 | 1.8120 | Yes |
| Restorability | 7 | -0.0789 | 0.2743 | 1.7610 | No |
| **Efficiency** | | | | | |
| I/O Utilization | 2 | 1.0000 | N/A | 2.1320 | Yes |
| Efficiency Compliance | 3 | 1.0000 | N/A | 1.9430 | Yes |
| **Usability** | | | | | |
| User Cancelability | 3 | -0.5774 | 1.7321 | 1.8600 | No |
| Usability Compliance | 7 | -0.4528 | 1.7593 | 1.7610 | No |
| **Maintainability** | | | | | |
| Data Logging | 12 | -0.0675 | 0.3314 | 1.7060 | No |
| **Portability** | | | | | |
| Software Operability | 13 | 0.1989 | 0.9942 | 1.7060 | No |
| Hardware Operability | 7 | 0.3919 | 1.5940 | 1.7460 | No |

Table B.15: Correlation of Integration/Test Skill Level 3 to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Tst Skill 3<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | 0.0923 | 0.6684 | 1.6726 | No |
| Functional Completeness | 28 | -0.2091 | 1.5717 | 1.6726 | No |
| Functional Coverage | 28 | 0.1773 | 1.3241 | 1.6726 | No |
| Specification Stability | 28 | -0.1715 | 1.2792 | 1.6726 | No |
| Accuracy | 9 | 0.5133 | 2.3922 | 1.7340 | Yes |
| Precision | 9 | -0.0221 | 0.0885 | 1.7340 | No |
| Data Exchangeability | 13 | 0.1163 | 0.5739 | 1.7060 | No |
| Interface Consistency | 9 | -0.4716 | 2.1394 | 1.7340 | Yes |
| Access Auditability | 7 | 0.2582 | 0.9258 | 1.7610 | No |
| Functional Compliance | 5 | -0.3430 | 1.0328 | 1.8120 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | -0.6873 | 3.5408 | 1.7460 | Yes |
| Incorrect Op Avoidance | 5 | -0.8385 | 4.3529 | 1.8120 | Yes |
| Restorability | 7 | -0.5960 | 2.5714 | 1.7610 | No |
| **Usability** | | | | | |
| Usability Compliance | 7 | -0.6455 | 2.9277 | 1.7610 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | 0.1953 | 0.9755 | 1.7060 | No |
| **Portability** | | | | | |
| Software Operability | 13 | -0.2343 | 1.1807 | 1.7060 | No |
| Hardware Operability | 7 | -0.1796 | 0.6831 | 1.7460 | No |

Table B.16: Correlation of Integration/Test Skill Level 4 to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Tst Skill 4<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | 0.1799 | 1.3189 | 1.6726 | No |
| Functional Completeness | 28 | 0.3493 | 2.7390 | 1.6726 | Yes |
| Functional Coverage | 28 | 0.3995 | 3.2022 | 1.6726 | Yes |
| Specification Stability | 28 | -0.2369 | 1.7918 | 1.6726 | Yes |
| Precision | 9 | -0.8047 | 5.4210 | 1.7340 | Yes |
| Data Exchangeability | 13 | -0.3886 | 2.0660 | 1.7060 | Yes |
| Interface Consistency | 9 | 0.3001 | 1.2585 | 1.7340 | No |
| Access Controllability | 8 | 0.2928 | 1.1456 | 1.7610 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.2037 | 0.7783 | 1.7460 | No |
| Restorability | 7 | 0.7024 | 3.4188 | 1.7610 | Yes |
| **Usability** | | | | | |
| Status Monitoring | 4 | 0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | 0.7303 | 3.7033 | 1.7610 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | 0.2855 | 1.4595 | 1.7060 | No |

Table B.17: Correlation of Specify Requirements Process to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Spec Reqs<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | 0.0952 | 0.6896 | 1.6726 | No |
| Functional Completeness | 28 | -0.1479 | 1.0991 | 1.6726 | No |
| Functional Coverage | 28 | 0.1745 | 1.3021 | 1.6726 | No |
| Specification Stability | 28 | 0.1570 | 1.1678 | 1.6726 | No |
| Access Controllability | 8 | -0.1429 | 0.5401 | 1.7460 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | -0.2037 | 0.7783 | 1.7460 | No |
| Restorability | 7 | -0.3311 | 1.2157 | 1.7610 | No |
| **Usability** | | | | | |
| Status Monitoring | 4 | -0.2500 | 0.7303 | 1.8120 | No |
| Usability Compliance | 7 | -0.2582 | 0.9258 | 1.7610 | No |
| **Maintainability** | | | | | |
| Data Logging | 12 | -0.1505 | 0.7457 | 1.7060 | No |

Table B.18: Correlation of Requirements Evaluation Process to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Eval Reqs<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | 0.4292 | 3.4266 | 1.6726 | Yes |
| Functional Completeness | 28 | 0.0083 | 0.0611 | 1.6726 | No |
| Functional Coverage | 28 | -0.1079 | 0.7975 | 1.6726 | No |
| Specification Stability | 28 | 0.0598 | 0.4400 | 1.6726 | No |
| Accuracy | 9 | 0.2144 | 0.8779 | 1.7340 | No |
| Precision | 9 | 0.0585 | 0.2345 | 1.7340 | No |
| Data Exchangeability | 13 | 0.5029 | 2.8501 | 1.7060 | Yes |
| Interface Consistency | 9 | -0.4610 | 2.0778 | 1.7340 | Yes |
| Access Auditability | 7 | -0.7303 | 3.7033 | 1.7610 | Yes |
| Access Controllability | 8 | 0.2182 | 0.8367 | 1.7460 | No |
| Data Corruption Prevention | 2 | 1.0000 | N/A | 2.1320 | Yes |
| Functional Compliance | 5 | -0.4082 | 1.2649 | 1.8120 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | -0.3246 | 1.2841 | 1.7460 | No |
| Incorrect Op Avoidance | 5 | -0.5590 | 1.9069 | 1.8120 | Yes |
| **Usability** | | | | | |
| User Cancelability | 3 | 0.3333 | 0.8660 | 1.8600 | No |
| Usability Compliance | 7 | -0.6455 | 2.9277 | 1.7610 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | -0.3301 | 1.7135 | 1.7060 | Yes |
| **Portability** | | | | | |
| Software Operability | 13 | 0.3705 | 1.9539 | 1.7060 | Yes |
| Hardware Operability | 7 | 0.0000 | N/A | 1.7460 | No |

Table B.19: Correlation of Update Requirements Process to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Update Reqs<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | 0.1605 | 1.1723 | 1.6726 | No |
| Functional Completeness | 28 | 0.3121 | 2.4141 | 1.6726 | Yes |
| Functional Coverage | 28 | 0.3570 | 2.8080 | 1.6726 | Yes |
| Specification Stability | 28 | -0.2447 | 1.8545 | 1.6726 | Yes |
| Precision | 9 | -0.8047 | 5.4210 | 1.7340 | No |
| Data Exchangeability | 13 | -0.2631 | 1.3358 | 1.7060 | No |
| Access Controllability | 8 | 0.2182 | 0.8367 | 1.7460 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.2037 | 0.7783 | 1.7460 | No |
| Restorability | 7 | 0.5130 | 2.0702 | 1.7610 | Yes |
| **Usability** | | | | | |
| Status Monitoring | 4 | 0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | 0.5477 | 2.2678 | 1.7610 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | 0.2855 | 1.4595 | 1.7060 | Yes |

Table B.20: Correlation of Develop Validation Criteria Process to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Val Critra<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | -0.1799 | 1.3189 | 1.6726 | No |
| Functional Completeness | 28 | -0.3493 | 2.7390 | 1.6726 | Yes |
| Functional Coverage | 28 | -0.3995 | 3.2022 | 1.6726 | Yes |
| Specification Stability | 28 | 0.2369 | 1.7918 | 1.6726 | Yes |
| Precision | 9 | 0.8047 | 5.4210 | 1.7340 | Yes |
| Data Exchangeability | 13 | 0.3886 | 2.0660 | 1.7060 | Yes |
| Interface Consistency | 9 | -0.3001 | 1.2585 | 1.7340 | No |
| Access Controllability | 8 | -0.2928 | 1.1456 | 1.7460 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | -0.2037 | 0.7783 | 1.7460 | No |
| Restorability | 7 | -0.7024 | 3.4188 | 1.7610 | Yes |
| **Usability** | | | | | |
| Status Monitoring | 4 | -0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | -0.7303 | 3.7033 | 1.7610 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | -0.2855 | 1.4595 | 1.7060 | No |

Table B.21: Correlation of Determine Environmental Impact Process to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Env Impact<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | 0.1799 | 1.3189 | 1.6726 | No |
| Functional Completeness | 28 | 0.3493 | 2.7390 | 1.6726 | Yes |
| Functional Coverage | 28 | 0.3995 | 3.2022 | 1.6726 | Yes |
| Specification Stability | 28 | -0.2369 | 1.7918 | 1.6726 | Yes |
| Precision | 9 | -0.8047 | 5.4210 | 1.7340 | Yes |
| Data Exchangeability | 13 | -0.3886 | 2.0660 | 1.7060 | Yes |
| Interface Consistency | 9 | 0.3001 | 1.2585 | 1.7340 | No |
| Access Controllability | 8 | 0.2928 | 1.1456 | 1.7460 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.2037 | 0.7783 | 1.7460 | No |
| Restorability | 7 | 0.7024 | 3.4188 | 1.7610 | Yes |
| **Usability** | | | | | |
| Status Monitoring | 4 | 0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | 0.7303 | 3.7033 | 1.7610 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | 0.2855 | 1.4595 | 1.7060 | No |

Table B.22: Correlation of Customer Evaluation of Requirements to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Cust Eval<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | -0.2342 | 1.7370 | 1.6726 | Yes |
| Functional Completeness | 28 | 0.6013 | 5.5303 | 1.6726 | Yes |
| Functional Coverage | 28 | 0.5314 | 4.6102 | 1.6726 | Yes |
| Specification Stability | 28 | -0.1073 | 0.7928 | 1.6726 | No |
| Accuracy | 9 | 0.2144 | 0.8779 | 1.7340 | No |
| Precision | 9 | -0.4534 | 2.0347 | 1.7340 | Yes |
| Data Exchangeability | 13 | -0.0593 | 0.2912 | 1.7060 | No |
| Interface Consistency | 9 | -0.2712 | 1.1269 | 1.7340 | No |
| Access Auditability | 7 | -0.7303 | 3.7033 | 1.7610 | Yes |
| Access Controllability | 8 | 0.4880 | 2.0917 | 1.7460 | Yes |
| Data Corruption Prevent | 2 | 1.0000 | N/A | 2.1320 | Yes |
| Functional Compliance | 5 | -0.4082 | 1.2649 | 1.8120 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | -0.1796 | 0.6831 | 1.7460 | No |
| Incorrect Op Avoidance | 5 | -0.5590 | 1.9069 | 1.8120 | Yes |
| Restorability | 7 | 0.7024 | 3.4188 | 1.7610 | Yes |
| **Usability** | | | | | |
| User Cancelability | 3 | 0.3333 | 0.8660 | 1.8600 | No |
| Status Monitoring | 4 | 0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | 0.7303 | 3.7033 | 1.7610 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | 0.0165 | 0.0808 | 1.7060 | No |
| **Portability** | | | | | |
| Software Operability | 13 | 0.1278 | 0.6313 | 1.7060 | Yes |
| Hardware Operability | 7 | 0.0000 | 0.0000 | 1.7460 | No |

Table B.23: Correlation of Communicate Requirements Process to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Comm Reqs<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | 0.1799 | 1.3189 | 1.6726 | No |
| Functional Completeness | 28 | 0.3493 | 2.7390 | 1.6726 | Yes |
| Functional Coverage | 28 | 0.3995 | 3.2022 | 1.6726 | Yes |
| Specification Stability | 28 | -0.2369 | 1.7918 | 1.6726 | Yes |
| Precision | 9 | -0.8047 | 5.4210 | 1.7340 | Yes |
| Data Exchangeability | 13 | -0.3886 | 2.0660 | 1.7060 | Yes |
| Interface Consistency | 9 | 0.3001 | 1.2585 | 1.7340 | No |
| Access Controllability | 8 | 0.2928 | 1.1456 | 1.7460 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.2037 | 0.7783 | 1.7460 | No |
| Restorability | 7 | 0.7024 | 3.4188 | 1.7610 | Yes |
| **Usability** | | | | | |
| Status Monitoring | 4 | 0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | 0.7303 | 3.7033 | 1.7610 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | 0.2855 | 1.4595 | 1.7060 | No |

Table B.24: Correlation of Develop Release Strategy Process to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Rel Strtgy<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | 0.1799 | 1.3189 | 1.6726 | No |
| Functional Completeness | 28 | 0.3493 | 2.7390 | 1.6726 | Yes |
| Functional Coverage | 28 | 0.3995 | 3.2022 | 1.6726 | Yes |
| Specification Stability | 28 | -0.2369 | 1.7918 | 1.6726 | Yes |
| Precision | 9 | -0.8047 | 5.4210 | 1.7340 | Yes |
| Data Exchangeability | 13 | -0.3886 | 2.0660 | 1.7060 | Yes |
| Interface Consistency | 9 | 0.3001 | 1.2585 | 1.7340 | No |
| Access Controllability | 8 | 0.2928 | 1.1456 | 1.7460 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.2037 | 0.7783 | 1.7460 | No |
| Restorability | 7 | 0.7024 | 3.4188 | 1.7610 | Yes |
| **Usability** | | | | | |
| Status Monitoring | 4 | 0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | 0.7303 | 3.7033 | 1.7610 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | 0.2855 | 1.4595 | 1.7060 | No |

Table B.25: Correlation of Develop Architecture Process to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Arch Design<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | -0.1404 | 1.0223 | 1.6726 | No |
| Functional Completeness | 28 | -0.2565 | 1.9501 | 1.6726 | Yes |
| Functional Coverage | 28 | -0.2957 | 2.2747 | 1.6726 | Yes |
| Specification Stability | 28 | 0.2101 | 1.5789 | 1.6726 | No |
| Precision | 9 | 0.8047 | 5.4210 | 1.7340 | Yes |
| Access Controllability | 8 | -0.2182 | 0.8367 | 1.7460 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | -0.2037 | 0.7783 | 1.7460 | No |
| Restorability | 7 | -0.5130 | 2.0702 | 1.7610 | Yes |
| **Usability** | | | | | |
| Status Monitoring | 4 | -0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | -0.4000 | 1.5119 | 1.7610 | No |
| **Maintainability** | | | | | |
| Data Logging | 12 | -0.2855 | 1.4595 | 1.7060 | Yes |

Table B.26: Correlation of Develop Detailed Design Process to Software Product Quality

| Software<br>Quality Variable | Samp.<br>Size | Det Design<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | -0.1404 | 1.0223 | 1.6726 | No |
| Functional Completeness | 28 | -0.2565 | 1.9501 | 1.6726 | Yes |
| Functional Coverage | 28 | -0.2957 | 2.2747 | 1.6726 | Yes |
| Specification Stability | 28 | 0.2101 | 1.5789 | 1.6726 | No |
| Precision | 9 | 0.8047 | 5.4210 | 1.7340 | Yes |
| Access Controllability | 8 | -0.2182 | 0.8367 | 1.7460 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | -0.2037 | 0.7783 | 1.7460 | No |
| Restorability | 7 | -0.5130 | 2.0702 | 1.7610 | Yes |
| **Usability** | | | | | |
| Status Monitoring | 4 | -0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | -0.4000 | 1.5119 | 1.7610 | No |
| **Maintainability** | | | | | |
| Data Logging | 12 | -0.2855 | 1.4595 | 1.7060 | Yes |

Table B.27: Correlation of Design Interfaces Process to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Design I/F<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | -0.1404 | 1.0223 | 1.6726 | No |
| Functional Completeness | 28 | -0.2565 | 1.9501 | 1.6726 | Yes |
| Functional Coverage | 28 | -0.2957 | 2.2747 | 1.6726 | Yes |
| Specification Stability | 28 | 0.2101 | 1.5789 | 1.6726 | No |
| Precision | 9 | 0.8047 | 5.4210 | 1.7340 | Yes |
| Access Controllability | 8 | -0.2182 | 0.8367 | 1.7460 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | -0.2037 | 0.7783 | 1.7460 | No |
| Restorability | 7 | -0.5130 | 2.0702 | 1.7610 | Yes |
| **Usability** | | | | | |
| Status Monitoring | 4 | -0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | -0.4000 | 1.5119 | 1.7610 | No |
| **Maintainability** | | | | | |
| Data Logging | 12 | -0.2855 | 1.4595 | 1.7060 | Yes |

Table B.28: Correlation of Verify Design Process to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Ver Design<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | -0.2342 | 1.7370 | 1.6726 | Yes |
| Functional Completeness | 28 | 0.6013 | 5.5303 | 1.6726 | Yes |
| Functional Coverage | 28 | 0.5314 | 4.6102 | 1.6726 | Yes |
| Specification Stability | 28 | -0.1073 | 0.7928 | 1.6726 | No |
| Accuracy | 9 | 0.2144 | 0.8779 | 1.7340 | No |
| Precision | 9 | -0.4534 | 2.0347 | 1.7340 | Yes |
| Data Exchangeability | 13 | -0.0593 | 0.2912 | 1.7060 | No |
| Interface Consistency | 9 | -0.2717 | 1.1269 | 1.7340 | No |
| Access Auditability | 7 | -0.7303 | 3.7033 | 1.7610 | Yes |
| Access Controllability | 8 | 0.4880 | 2.0917 | 1.7460 | Yes |
| Data Corruption Prevent | 2 | 1.0000 | N/A | 2.1320 | Yes |
| Functional Compliance | 5 | -0.4082 | 1.2649 | 1.8120 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | -0.1796 | 0.6831 | 1.7460 | No |
| Incorrect Op Avoidance | 5 | -0.5590 | 1.9069 | 1.8120 | Yes |
| Restorability | 7 | 0.7024 | 3.4188 | 1.7610 | Yes |
| **Usability** | | | | | |
| User Cancelability | 3 | 0.3333 | 0.8660 | 1.8600 | No |
| Status Monitoring | 4 | 0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | 0.7303 | 3.7033 | 1.7610 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | 0.0165 | 0.0808 | 1.7060 | No |
| **Portability** | | | | | |
| Software Operability | 13 | 0.1278 | 0.6313 | 1.7060 | No |
| Hardware Operability | 7 | 0.0000 | 0.0000 | 1.7460 | No |

Table B.29: Correlation of Design Traceability Process to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Des Trace<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | -0.4292 | 3.4266 | 1.6726 | Yes |
| Functional Completeness | 28 | 0.0083 | 0.0611 | 1.6726 | No |
| Functional Coverage | 28 | -0.1079 | 0.7975 | 1.6726 | No |
| Specification Stability | 28 | 0.0598 | 0.4400 | 1.6726 | No |
| Accuracy | 9 | 0.2144 | 0.8779 | 1.7340 | No |
| Precision | 9 | 0.0585 | 0.2345 | 1.7340 | No |
| Data Exchangeability | 13 | 0.5029 | 2.8501 | 1.7060 | Yes |
| Interface Consistency | 9 | -0.4610 | 2.0778 | 1.7340 | Yes |
| Access Auditability | 7 | -0.7303 | 3.7033 | 1.7610 | Yes |
| Access Controllability | 8 | 0.2182 | 0.8367 | 1.7460 | No |
| Data Corruption Prevent | 2 | 1.0000 | N/A | 2.1320 | Yes |
| Functional Compliance | 5 | -0.4082 | 1.2649 | 1.8120 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | -0.3246 | 1.2841 | 1.7460 | No |
| Incorrect Op Avoidance | 5 | -0.5590 | 1.9069 | 1.8120 | Yes |
| **Usability** | | | | | |
| User Cancelability | 3 | 0.3333 | 0.8660 | 1.8600 | No |
| Usability Compliance | 7 | -0.6455 | 2.9277 | 1.7610 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | -0.3301 | 1.7135 | 1.7060 | Yes |
| **Portability** | | | | | |
| Software Operability | 13 | 0.3705 | 1.9539 | 1.7060 | Yes |
| Hardware Operability | 7 | 0.0000 | 0.0000 | 1.7460 | No |

Table B.30: Correlation of Develop Unit Test Process to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Dev Unit Tst<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | 0.1799 | 1.3189 | 1.6726 | No |
| Functional Completeness | 28 | 0.3493 | 2.7390 | 1.6726 | Yes |
| Functional Coverage | 28 | 0.3995 | 3.2022 | 1.6726 | Yes |
| Specification Stability | 28 | -0.2369 | 1.7918 | 1.6726 | Yes |
| Precision | 9 | -0.8047 | 5.4210 | 1.7340 | Yes |
| Data Exchangeability | 13 | -0.3886 | 2.0660 | 1.7060 | Yes |
| Interface Consistency | 9 | 0.3001 | 1.2585 | 1.7340 | No |
| Access Controllability | 8 | 0.2928 | 1.1456 | 1.7460 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.2037 | 0.7783 | 1.7460 | No |
| Restorability | 7 | 0.7024 | 3.4188 | 1.7610 | Yes |
| **Usability** | | | | | |
| Status Monitoring | 4 | 0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | 0.7303 | 3.7033 | 1.7610 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | 0.2855 | 1.4595 | 1.7060 | No |

Table B.31: Correlation of Verify Software Units Process to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Verify Unit<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | -0.3319 | 2.5373 | 1.6726 | Yes |
| Functional Completeness | 28 | 0.1441 | 1.0701 | 1.6726 | No |
| Functional Coverage | 28 | 0.0531 | 0.3906 | 1.6726 | No |
| Specification Stability | 28 | -0.0494 | 0.7928 | 1.6726 | No |
| Precision | 9 | -0.0357 | 0.2345 | 1.7340 | No |
| Data Exchangeability | 13 | 0.5029 | 2.8501 | 1.7060 | Yes |
| Interface Consistency | 9 | -0.4610 | 2.0778 | 1.7340 | Yes |
| Access Auditability | 7 | -0.7303 | 3.7033 | 1.7610 | Yes |
| Access Controllability | 8 | 0.2928 | 1.1456 | 1.7460 | No |
| Data Corruption Prevent | 2 | 1.0000 | N/A | 2.1320 | Yes |
| Functional Compliance | 5 | -0.4082 | 1.2649 | 1.8120 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | -0.1796 | 0.6831 | 1.7460 | No |
| Incorrect Op Avoidance | 5 | -0.5590 | 1.9069 | 1.8120 | Yes |
| Restorability | 7 | 0.5130 | 2.0702 | 1.7610 | Yes |
| **Usability** | | | | | |
| User Cancelability | 3 | 0.3333 | 0.8660 | 1.8600 | No |
| Status Monitoring | 4 | 0.4082 | 1.2649 | 1.8120 | No |
| Usability Compliance | 7 | -0.3000 | 1.0894 | 1.7610 | No |
| **Maintainability** | | | | | |
| Data Logging | 12 | -0.1448 | 0.7169 | 1.7060 | No |
| **Portability** | | | | | |
| Software Operability | 13 | 0.3705 | 1.9539 | 1.7060 | Yes |
| Hardware Operability | 7 | 0.0000 | 0.0000 | 1.7460 | No |

Table B.32: Correlation of Unit Traceability Process to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Unit Trace<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | -0.4292 | 3.4266 | 1.6726 | Yes |
| Functional Completeness | 28 | 0.0083 | 0.0611 | 1.6726 | No |
| Functional Coverage | 28 | -0.1079 | 0.7975 | 1.6726 | No |
| Specification Stability | 28 | 0.0598 | 0.4400 | 1.6726 | No |
| Accuracy | 9 | 0.2144 | 0.8799 | 1.7340 | No |
| Precision | 9 | 0.0585 | 0.2345 | 1.7340 | No |
| Data Exchangeability | 13 | 0.5029 | 2.8501 | 1.7060 | Yes |
| Interface Consistency | 9 | -0.4610 | 2.0778 | 1.7340 | Yes |
| Access Auditability | 7 | -0.7303 | 3.7033 | 1.7610 | Yes |
| Access Controllability | 8 | 0.2182 | 0.8367 | 1.7460 | No |
| Data Corruption Prevent | 2 | 1.0000 | N/A | 2.1320 | Yes |
| Functional Compliance | 5 | -0.4082 | 1.2649 | 1.8120 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | -0.3246 | 1.2841 | 1.7460 | No |
| Incorrect Op Avoidance | 5 | -0.5590 | 1.9069 | 1.8120 | Yes |
| **Usability** | | | | | |
| User Cancelability | 3 | 0.3333 | 0.8660 | 1.8600 | No |
| Usability Compliance | 7 | -0.6455 | 2.9277 | 1.7610 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | -0.3301 | 1.7135 | 1.7060 | Yes |
| **Portability** | | | | | |
| Software Operability | 13 | 0.3705 | 1.9535 | 1.7060 | Yes |
| Hardware Operability | 7 | 0.0000 | 0.0000 | 1.7460 | No |

Table B.33: Correlation of Integration Strategy Process to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Integ Stgy<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | 0.1799 | 1.3189 | 1.6726 | No |
| Functional Completeness | 28 | 0.3493 | 2.7390 | 1.6726 | Yes |
| Functional Coverage | 28 | 0.3995 | 3.2022 | 1.6726 | Yes |
| Specification Stability | 28 | -0.2369 | 1.7918 | 1.6726 | Yes |
| Precision | 9 | -0.8047 | 5.4210 | 1.7340 | Yes |
| Data Exchangeability | 13 | -0.3886 | 2.0660 | 1.7060 | Yes |
| Interface Consistency | 9 | 0.3001 | 1.2585 | 1.7340 | No |
| Access Controllability | 8 | 0.2928 | 1.1456 | 1.7460 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.2037 | 0.7783 | 1.7460 | No |
| Restorability | 7 | 0.7024 | 3.4188 | 1.7610 | Yes |
| **Usability** | | | | | |
| Status Monitoring | 4 | 0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | 0.7303 | 3.7033 | 1.7610 | Yes |

Table B.34: Correlation of Regression Test Strategy Process to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Regrss Stgy<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | 0.1799 | 1.3189 | 1.6726 | No |
| Functional Completeness | 28 | 0.3493 | 2.7390 | 1.6726 | Yes |
| Functional Coverage | 28 | 0.3995 | 3.2022 | 1.6726 | Yes |
| Specification Stability | 28 | -0.2369 | 1.7918 | 1.6726 | Yes |
| Precision | 9 | -0.8047 | 5.4210 | 1.7340 | Yes |
| Data Exchangeability | 13 | -0.3886 | 2.0660 | 1.7060 | Yes |
| Interface Consistency | 9 | 0.3001 | 1.2585 | 1.7340 | No |
| Access Controllability | 8 | 0.2928 | 1.1456 | 1.7460 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.2037 | 0.7783 | 1.7460 | No |
| Restorability | 7 | 0.7024 | 3.4188 | 1.7610 | Yes |
| **Usability** | | | | | |
| Status Monitoring | 4 | 0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | 0.7303 | 3.7033 | 1.7610 | Yes |

Table B.35: Correlation of Regression Integration Tests Process to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Reg Int Tst<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | 0.1799 | 1.3189 | 1.6726 | No |
| Functional Completeness | 28 | 0.3493 | 2.7390 | 1.6726 | Yes |
| Functional Coverage | 28 | 0.3995 | 3.2022 | 1.6726 | Yes |
| Specification Stability | 28 | -0.2369 | 1.7918 | 1.6726 | Yes |
| Precision | 9 | -0.8047 | 5.4210 | 1.7340 | Yes |
| Data Exchangeability | 13 | -0.3886 | 2.0660 | 1.7060 | Yes |
| Interface Consistency | 9 | 0.3001 | 1.2585 | 1.7340 | No |
| Access Controllability | 8 | 0.2928 | 1.1456 | 1.7460 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.2037 | 0.7783 | 1.7460 | No |
| Restorability | 7 | 0.7024 | 3.4188 | 1.7610 | Yes |
| **Usability** | | | | | |
| Status Monitoring | 4 | 0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | 0.7303 | 3.7033 | 1.7610 | Yes |

Table B.36: Correlation of Develop System Test Process to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Dev Sys Tst<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | -0.1799 | 1.3189 | 1.6726 | No |
| Functional Completeness | 28 | -0.3493 | 2.7390 | 1.6726 | Yes |
| Functional Coverage | 28 | -0.3995 | 3.2022 | 1.6726 | Yes |
| Specification Stability | 28 | 0.2369 | 1.7918 | 1.6726 | Yes |
| Precision | 9 | 0.8047 | 5.4210 | 1.7340 | Yes |
| Data Exchangeability | 13 | 0.3886 | 2.0660 | 1.7060 | Yes |
| Interface Consistency | 9 | -0.3001 | 1.2585 | 1.7340 | No |
| Access Controllability | 8 | -0.2928 | 1.1456 | 1.7460 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | -0.2037 | 0.7783 | 1.7460 | No |
| Restorability | 7 | -0.7024 | 3.4188 | 1.7610 | Yes |
| **Usability** | | | | | |
| Status Monitoring | 4 | -0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | -0.7303 | 3.7033 | 1.7610 | Yes |

Table B.37: Correlation of Regression Test System Process to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Reg Tst Sys<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | 0.1799 | 1.3189 | 1.6726 | No |
| Functional Completeness | 28 | 0.3493 | 2.7390 | 1.6726 | Yes |
| Functional Coverage | 28 | 0.3995 | 3.2022 | 1.6726 | Yes |
| Specification Stability | 28 | -0.2369 | 1.7918 | 1.6726 | Yes |
| Precision | 9 | -0.8047 | 5.4210 | 1.7340 | Yes |
| Data Exchangeability | 13 | -0.3886 | 2.0660 | 1.7060 | Yes |
| Interface Consistency | 9 | 0.3001 | 1.2585 | 1.7340 | No |
| Access Controllability | 8 | 0.2928 | 1.1456 | 1.7460 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.2037 | 0.7783 | 1.7460 | No |
| Restorability | 7 | 0.7024 | 3.4188 | 1.7610 | Yes |
| **Usability** | | | | | |
| Status Monitoring | 4 | 0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | 0.7303 | 3.7033 | 1.7610 | Yes |

Table B.38: Correlation of Requirements Coverage to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Coverage<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Usability** | | | | | |
| User Cancelability | 3 | -0.3333 | 0.8660 | 1.8600 | No |
| Status Monitoring | 4 | -0.2500 | 0.7303 | 1.8120 | No |
| **Maintainability** | | | | | |
| Data Logging | 12 | -0.1505 | 0.7457 | 1.7060 | No |
| **Portability** | | | | | |
| Software Operability | 13 | -0.2417 | 1.2201 | 1.7060 | No |
| Hardware Operability | 7 | -0.3780 | 1.5275 | 1.7460 | No |

Table B.39: Correlation of Implementation Prevalence to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Prevalence<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Precision | 9 | -0.7686 | 2.4030 | 1.9430 | Yes |
| Data Exchangeability | 13 | 0.8278 | 5.1123 | 1.7610 | Yes |
| Interface Consistency | 9 | 0.5340 | 1.7863 | 1.8120 | No |
| Functional Compliance | 5 | 0.9582 | 9.4789 | 1.8120 | Yes |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.2429 | 0.6133 | 1.8600 | No |
| Restorability | 7 | 0.0258 | 0.0731 | 1.8120 | No |
| **Usability** | | | | | |
| User Cancelability | 3 | 0.0000 | 0.0000 | 1.8600 | No |
| Usability Compliance | 7 | -0.1825 | 0.5869 | 1.7820 | No |
| **Maintainability** | | | | | |
| Data Logging | 12 | 0.2912 | 0.9627 | 1.7820 | No |

Table B.40: Correlation of Depth of Inheritance Tree to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | DIT<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | -0.1510 | 1.1012 | 1.6726 | No |
| Functional Completeness | 28 | -0.2440 | 1.8490 | 1.6726 | Yes |
| Functional Coverage | 28 | -0.2769 | 2.1177 | 1.6726 | Yes |
| Specification Stability | 28 | -0.4259 | 3.4588 | 1.6726 | Yes |
| Accuracy | 9 | -0.1286 | 0.5188 | 1.7340 | No |
| Precision | 9 | -0.1578 | 0.6393 | 1.7340 | No |
| Data Exchangeability | 13 | 0.1474 | 0.7302 | 1.7060 | No |
| Interface Consistency | 9 | 0.1966 | 0.8019 | 1.7340 | No |
| Access Auditability | 7 | 0.4282 | 1.6413 | 1.7610 | No |
| Access Controllability | 8 | 0.2607 | 1.2585 | 1.7460 | No |
| Data Encryption | 2 | 1.0000 | N/A | 2.1320 | Yes |
| Functional Compliance | 5 | 0.5145 | 1.6971 | 1.8120 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.5357 | 2.3735 | 1.7460 | Yes |
| Incorrect Op Avoidance | 5 | 0.6578 | 2.4702 | 1.8120 | Yes |
| Restorability | 7 | 0.3035 | 1.1034 | 1.7610 | No |
| **Efficiency** | | | | | |
| Efficiency Compliance | 3 | 0.1890 | 0.3849 | 1.9430 | No |
| **Usability** | | | | | |
| User Cancelability | 3 | 0.1741 | 0.4330 | 1.8600 | No |
| Status Monitoring | 4 | 0.6124 | 2.1909 | 1.8120 | Yes |
| Usability Compliance | 7 | -0.4000 | 1.5119 | 1.7610 | No |
| **Maintainability** | | | | | |
| Data Logging | 12 | 0.2593 | 1.3154 | 1.7060 | No |
| **Portability** | | | | | |
| Software Operability | 13 | 0.1674 | 0.8320 | 1.7060 | No |
| Hardware Operability | 7 | 0.0000 | 0.0000 | 1.7460 | No |

Table B.41: Correlation of Design Expansion to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Des Expand<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | 0.0183 | 0.1317 | 1.6726 | No |
| Functional Completeness | 28 | -0.1350 | 1.0010 | 1.6726 | No |
| Functional Coverage | 28 | -0.1310 | 0.9712 | 1.6726 | No |
| Specification Stability | 28 | -0.0186 | 0.1366 | 1.6726 | No |
| Accuracy | 9 | 0.3673 | 1.5798 | 1.7340 | No |
| Precision | 9 | 0.2800 | 1.1667 | 1.7340 | No |
| Data Exchangeability | 13 | 0.3555 | 1.8630 | 1.7060 | Yes |
| Interface Consistency | 9 | 0.1034 | 0.4158 | 1.7340 | No |
| Access Auditability | 7 | -0.0537 | 0.1863 | 1.7610 | No |
| Access Controllability | 8 | -0.3263 | 1.2917 | 1.7460 | No |
| Functional Compliance | 5 | -0.5045 | 1.6527 | 1.8120 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.1621 | 0.6147 | 1.7460 | No |
| Incorrect Op Avoidance | 5 | -0.2818 | 0.8308 | 1.8120 | No |
| Restorability | 7 | -0.4626 | 1.8074 | 1.7610 | Yes |
| **Efficiency** | | | | | |
| I/O Utilization | 2 | 1.0000 | N/A | 1.7340 | Yes |
| Efficiency Compliance | 3 | -0.0822 | 0.1650 | 1.9430 | No |
| **Usability** | | | | | |
| User Cancelability | 3 | -0.5659 | 1.6813 | 1.8600 | No |
| Status Monitoring | 4 | -0.9824 | 1.48E1 | 1.8120 | Yes |
| Usability Compliance | 7 | -0.7065 | 3.4581 | 1.7610 | Yes |
| **Maintainability** | | | | | |
| Data Logging | 12 | 0.1812 | 0.9027 | 1.7060 | No |
| **Portability** | | | | | |
| Software Operability | 13 | -0.4288 | 2.3250 | 1.7060 | Yes |
| Hardware Operability | 7 | -0.2692 | 1.0458 | 1.7460 | No |

Table B.42: Correlation of Interface Format Expansion to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | I/F Fmt Exp<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | -0.2760 | 1.7227 | 1.6864 | Yes |
| Functional Completeness | 28 | 0.1395 | 0.8450 | 1.6864 | No |
| Functional Coverage | 28 | 0.0858 | 0.5170 | 1.6864 | No |
| Specification Stability | 28 | 0.0657 | 0.3949 | 1.6864 | No |
| Accuracy | 9 | -0.3059 | 0.9087 | 1.8120 | No |
| Precision | 9 | 0.6752 | 2.2419 | 1.8600 | Yes |
| Data Exchangeability | 13 | 0.3927 | 1.7083 | 1.7340 | No |
| Interface Consistency | 9 | 0.0230 | 0.0796 | 1.7610 | No |
| Access Auditability | 7 | -0.7877 | 3.1322 | 1.8600 | Yes |
| Access Controllability | 8 | -0.0916 | 0.2909 | 1.7820 | No |
| Functional Compliance | 5 | 0.1147 | 0.2309 | 1.9430 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | -0.2500 | 0.7303 | 1.8120 | No |
| Restorability | 7 | 0.4042 | 1.0825 | 1.8600 | No |
| **Efficiency** | | | | | |
| I/O Utilization | 2 | 1.0000 | N/A | 2.1320 | Yes |
| Efficiency Compliance | 3 | 1.0000 | N/A | 2.1320 | Yes |
| **Usability** | | | | | |
| Status Monitoring | 4 | -0.3503 | 1.0579 | 1.8120 | No |
| Usability Compliance | 7 | -0.0376 | 0.1188 | 1.7820 | No |
| **Maintainability** | | | | | |
| Data Logging | 12 | -0.5138 | 2.5406 | 1.7250 | Yes |
| **Portability** | | | | | |
| Software Operability | 13 | -0.1226 | 0.4623 | 1.7460 | No |
| Hardware Operability | 7 | 0.0000 | 0.0000 | 1.8120 | No |

Table B.43: Correlation of Interface Protocol Expansion to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | I/F Prt Exp<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | -0.3776 | 2.3776 | 1.6888 | Yes |
| Functional Completeness | 28 | -0.0167 | 0.0976 | 1.6888 | No |
| Functional Coverage | 28 | -0.1786 | 1.0585 | 1.6888 | No |
| Specification Stability | 28 | 0.0785 | 0.4593 | 1.6888 | No |
| Accuracy | 9 | -0.5270 | 1.7541 | 1.8120 | No |
| Precision | 9 | 0.5222 | 1.5000 | 1.8600 | No |
| Data Exchangeability | 13 | 0.3977 | 1.5016 | 1.7610 | No |
| Interface Consistency | 9 | -0.4894 | 2.0996 | 1.7460 | Yes |
| Access Auditability | 7 | -0.5000 | 1.1547 | 1.9430 | No |
| Functional Compliance | 5 | 0.2928 | 0.7500 | 1.8600 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.2500 | 0.7303 | 1.8120 | No |
| **Efficiency** | | | | | |
| I/O Utilization | 2 | 1.0000 | N/A | 2.1320 | Yes |
| Efficiency Compliance | 3 | 1.0000 | N/A | 2.1320 | Yes |
| **Usability** | | | | | |
| Usability Compliance | 7 | 0.2000 | 0.6455 | 1.7820 | No |
| **Maintainability** | | | | | |
| Data Logging | 12 | -0.5185 | 2.2687 | 1.7460 | Yes |
| **Portability** | | | | | |
| Software Operability | 13 | 0.0000 | 0.0000 | 1.7610 | No |
| Hardware Operability | 7 | 0.2582 | 0.6547 | 1.8600 | No |

Table B.44: Correlation of Requirements Volatility to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Req Volaty<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | 0.1846 | 1.3545 | 1.6726 | No |
| Functional Completeness | 28 | 0.0143 | 0.1052 | 1.6726 | No |
| Functional Coverage | 28 | 0.0645 | 0.4750 | 1.6726 | No |
| Specification Stability | 28 | 0.9977 | 1.09E2 | 1.6726 | Yes |
| Accuracy | 9 | -0.6013 | 3.0102 | 1.7340 | Yes |
| Precision | 9 | -0.0357 | 0.1429 | 1.7340 | No |
| Data Exchangeability | 13 | -0.1996 | 0.9979 | 1.7060 | No |
| Interface Consistency | 9 | -0.2748 | 1.1431 | 1.7340 | No |
| Access Auditability | 7 | 0.3171 | 1.1582 | 1.7610 | No |
| Access Controllability | 8 | 0.1837 | 0.6992 | 1.7460 | No |
| Functional Compliance | 5 | 0.0963 | 0.2736 | 1.8120 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | -0.2236 | 0.8583 | 1.7460 | No |
| Incorrect Op Avoidance | 5 | 0.0129 | 0.0365 | 1.8120 | No |
| Restorability | 7 | 0.0862 | 0.2998 | 1.7610 | No |
| **Efficiency** | | | | | |
| I/O Utilization | 2 | 1.0000 | N/A | 2.1320 | Yes |
| Efficiency Compliance | 3 | 0.7450 | 2.2336 | 1.9430 | Yes |
| **Usability** | | | | | |
| User Cancelability | 3 | 0.2196 | 0.5512 | 1.8600 | No |
| Status Monitoring | 4 | 0.2500 | 0.7303 | 1.8120 | No |
| Usability Compliance | 7 | 0.3495 | 1.2921 | 1.7610 | No |
| **Maintainability** | | | | | |
| Data Logging | 12 | -0.5381 | 3.1278 | 1.7060 | Yes |
| **Portability** | | | | | |
| Software Operability | 13 | 0.1717 | 0.8538 | 1.7060 | No |
| Hardware Operability | 7 | -0.4340 | 1.8026 | 1.7460 | Yes |

Table B.45: Correlation of Design Volatility to Software Quality

| Software<br>Quality Variable | Samp.<br>Size | Des Volaty<br>Correlation | Test<br>Stat(t) | Critical<br>Value(cv) | Significant?<br>(t > cv) |
|---|---|---|---|---|---|
| **Suitability** | | | | | |
| Functional Adequacy | 28 | -0.1077 | 0.7809 | 1.6726 | No |
| Functional Completeness | 28 | 0.0237 | 0.1743 | 1.6726 | No |
| Functional Coverage | 28 | -0.0031 | 0.0231 | 1.6726 | No |
| Specification Stability | 28 | 0.1607 | 1.1964 | 1.6726 | No |
| Accuracy | 9 | -0.5223 | 2.4499 | 1.7340 | Yes |
| Precision | 9 | 0.3527 | 1.5030 | 1.7340 | No |
| Data Exchangeability | 13 | 0.0056 | 0.0272 | 1.7060 | No |
| Interface Consistency | 9 | 0.0094 | 0.0378 | 1.7340 | No |
| Access Auditability | 7 | -0.6455 | 2.9277 | 1.7610 | Yes |
| Access Controllability | 8 | 0.1429 | 0.5401 | 1.7460 | No |
| Functional Compliance | 5 | 0.1103 | 0.3140 | 1.8120 | No |
| **Reliability** | | | | | |
| Failure Avoidance | 8 | 0.2903 | 1.1352 | 1.7460 | No |
| **Efficiency** | | | | | |
| I/O Utilization | 2 | 1.0000 | N/A | 2.1320 | Yes |
| Efficiency Compliance | 3 | 0.3054 | 0.6415 | 1.9430 | No |
| **Usability** | | | | | |
| User Cancelability | 3 | 0.3333 | 0.8660 | 1.8600 | No |
| Usability Compliance | 7 | 0.2582 | 0.9258 | 1.7610 | No |
| **Maintainability** | | | | | |
| Data Logging | 12 | -0.8736 | 8.7955 | 1.7060 | Yes |
| **Portability** | | | | | |
| Software Operability | 13 | 0.2190 | 1.0996 | 1.7060 | No |
| Hardware Operability | 7 | -0.5044 | 2.1857 | 1.7460 | Yes |

# APPENDIX C

# RAW DATA TABLES

Table C.1: Requirements and Design Skill Data

| | Requirements | | | | | Design | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Proj. | Team | Num | Num | Num | Num | Team | Num | Num | Num | Num |
| ID | Size | Skill1 | Skill2 | Skill3 | Skill4 | Size | Skill1 | Skill2 | Skill3 | Skill4 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 2 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 3 | 2 | 0 | 2 | 0 | 0 | 2 | 0 | 2 | 0 | 0 |
| 4 | 2 | 0 | 2 | 0 | 0 | 2 | 1 | 1 | 0 | 0 |
| 5 | 2 | 1 | 1 | 0 | 0 | 2 | 0 | 2 | 0 | 0 |
| 6 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 7 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 8 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 9 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 10 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 11 | 3 | 1 | 2 | 0 | 0 | 3 | 1 | 2 | 0 | 0 |
| 12 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 13 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 14 | 2 | 0 | 1 | 1 | 0 | 2 | 0 | 2 | 0 | 0 |
| 15 | 3 | 1 | 1 | 1 | 0 | 3 | 1 | 1 | 0 | 1 |
| 16 | 3 | 0 | 3 | 0 | 0 | 3 | 1 | 1 | 1 | 0 |
| 17 | 3 | 0 | 3 | 0 | 0 | 3 | 2 | 1 | 0 | 0 |
| 18 | 3 | 0 | 2 | 1 | 0 | 3 | 0 | 3 | 0 | 0 |
| 19 | 3 | 0 | 3 | 0 | 0 | 3 | 0 | 3 | 0 | 0 |
| 20 | 3 | 1 | 2 | 0 | 0 | 3 | 0 | 3 | 0 | 0 |
| 21 | 3 | 0 | 3 | 0 | 0 | 3 | 0 | 3 | 0 | 0 |
| 22 | 4 | 1 | 3 | 0 | 0 | 4 | 1 | 3 | 0 | 0 |
| 23 | 4 | 0 | 1 | 2 | 1 | 4 | 0 | 1 | 2 | 1 |
| 24 | 4 | 0 | 1 | 2 | 1 | 4 | 0 | 1 | 2 | 1 |
| 25 | 4 | 0 | 1 | 2 | 1 | 4 | 0 | 1 | 2 | 1 |
| 26 | 4 | 0 | 1 | 2 | 1 | 4 | 0 | 1 | 2 | 1 |
| 27 | 4 | 0 | 1 | 2 | 1 | 4 | 0 | 1 | 2 | 1 |
| 28 | 4 | 0 | 1 | 2 | 1 | 4 | 0 | 1 | 2 | 1 |

Table C.2: Implementation and Integration/Test Skill Data

| | Implementation | | | | | Integration/Test | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Proj. ID | Team Size | Num Skill1 | Num Skill2 | Num Skill3 | Num Skill4 | Team Size | Num Skill1 | Num Skill2 | Num Skill3 | Num Skill4 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 2 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 3 | 2 | 0 | 2 | 0 | 0 | 2 | 0 | 2 | 0 | 0 |
| 4 | 2 | 0 | 2 | 0 | 0 | 2 | 1 | 0 | 1 | 0 |
| 5 | 2 | 0 | 2 | 0 | 0 | 2 | 1 | 1 | 0 | 0 |
| 6 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 7 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 8 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 9 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 10 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 11 | 3 | 0 | 2 | 1 | 0 | 3 | 1 | 2 | 0 | 0 |
| 12 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 13 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 14 | 2 | 0 | 1 | 1 | 0 | 2 | 1 | 1 | 0 | 0 |
| 15 | 3 | 1 | 1 | 0 | 1 | 3 | 1 | 1 | 1 | 0 |
| 16 | 3 | 0 | 1 | 2 | 0 | 3 | 1 | 1 | 1 | 0 |
| 17 | 3 | 0 | 3 | 0 | 0 | 3 | 0 | 3 | 0 | 0 |
| 18 | 3 | 0 | 1 | 1 | 1 | 3 | 0 | 3 | 0 | 0 |
| 19 | 3 | 0 | 3 | 0 | 0 | 3 | 0 | 3 | 0 | 0 |
| 20 | 3 | 0 | 2 | 1 | 0 | 3 | 0 | 2 | 1 | 0 |
| 21 | 3 | 0 | 3 | 0 | 0 | 3 | 0 | 3 | 0 | 0 |
| 22 | 4 | 0 | 3 | 1 | 0 | 4 | 2 | 2 | 0 | 0 |
| 23 | 4 | 0 | 0 | 1 | 3 | 1 | 0 | 0 | 0 | 1 |
| 24 | 4 | 0 | 0 | 1 | 3 | 1 | 0 | 0 | 0 | 1 |
| 25 | 4 | 0 | 0 | 1 | 3 | 1 | 0 | 0 | 0 | 1 |
| 26 | 4 | 0 | 0 | 1 | 3 | 1 | 0 | 0 | 0 | 1 |
| 27 | 4 | 0 | 0 | 1 | 3 | 1 | 0 | 0 | 0 | 1 |
| 28 | 4 | 0 | 0 | 1 | 3 | 1 | 0 | 0 | 0 | 1 |

Table C.3: Requirements Process Data

| Proj. | Specify | Rel | Val | Env | Updt | Comm | Eval | Eval |
|---|---|---|---|---|---|---|---|---|
| ID | Reqs | Stgy | Crit | Impact | Reqs | Reqs | Cust | Reqs |
| 1 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 1.0 |
| 2 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 3 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 1.0 |
| 4 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 5 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 6 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 7 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 8 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 1.0 |
| 9 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 1.0 |
| 10 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 11 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 1.0 |
| 12 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 13 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 14 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 15 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 1.0 |
| 16 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 |
| 17 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 1.0 |
| 18 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 1.0 |
| 19 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 20 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 |
| 21 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 22 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 1.0 |
| 23 | 1.0 | 1.0 | 0.0 | 1.0 | 0.0 | 1.0 | 1.0 | 0.0 |
| 24 | 1.0 | 1.0 | 0.0 | 1.0 | 1.0 | 1.0 | 1.0 | 0.0 |
| 25 | 1.0 | 1.0 | 0.0 | 1.0 | 1.0 | 1.0 | 1.0 | 0.0 |
| 26 | 0.0 | 1.0 | 0.0 | 1.0 | 1.0 | 1.0 | 1.0 | 0.0 |
| 27 | 0.0 | 1.0 | 0.0 | 1.0 | 1.0 | 1.0 | 1.0 | 0.0 |
| 28 | 1.0 | 1.0 | 0.0 | 1.0 | 1.0 | 1.0 | 1.0 | 0.0 |

Table C.4: Design Process Data

| Proj. ID | Develop Archtctr | Develop Det Dsgn | Dsgn I/Fs | Vrfy Dsgn | Dsgn Trace |
|---|---|---|---|---|---|
| 1 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| 2 | 1.0 | 1.0 | 1.0 | 0.0 | 0.0 |
| 3 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| 4 | 1.0 | 1.0 | 1.0 | 0.0 | 0.0 |
| 5 | 1.0 | 1.0 | 1.0 | 0.0 | 0.0 |
| 6 | 1.0 | 1.0 | 1.0 | 0.0 | 0.0 |
| 7 | 1.0 | 1.0 | 1.0 | 0.0 | 0.0 |
| 8 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| 9 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| 10 | 1.0 | 1.0 | 1.0 | 0.0 | 0.0 |
| 11 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| 12 | 1.0 | 1.0 | 1.0 | 0.0 | 0.0 |
| 13 | 1.0 | 1.0 | 1.0 | 0.0 | 0.0 |
| 14 | 1.0 | 1.0 | 1.0 | 0.0 | 0.0 |
| 15 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| 16 | 1.0 | 1.0 | 1.0 | 1.0 | 0.0 |
| 17 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| 18 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| 19 | 1.0 | 1.0 | 1.0 | 0.0 | 0.0 |
| 20 | 1.0 | 1.0 | 1.0 | 0.0 | 1.0 |
| 21 | 1.0 | 1.0 | 1.0 | 0.0 | 0.0 |
| 22 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| 23 | 1.0 | 1.0 | 1.0 | 1.0 | 0.0 |
| 24 | 1.0 | 1.0 | 1.0 | 1.0 | 0.0 |
| 25 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 |
| 26 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 |
| 27 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 |
| 28 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 |

Table C.5: Implementation Process Data

| Proj. ID | Develop Units | Develop Unit Tsts | Vrfy Unit | Unit Trace |
|---|---|---|---|---|
| 1 | 1.0 | 0.0 | 1.0 | 1.0 |
| 2 | 1.0 | 0.0 | 0.0 | 0.0 |
| 3 | 1.0 | 0.0 | 1.0 | 1.0 |
| 4 | 1.0 | 0.0 | 0.0 | 0.0 |
| 5 | 1.0 | 0.0 | 0.0 | 0.0 |
| 6 | 1.0 | 0.0 | 0.0 | 0.0 |
| 7 | 1.0 | 0.0 | 0.0 | 0.0 |
| 8 | 1.0 | 0.0 | 1.0 | 1.0 |
| 9 | 1.0 | 0.0 | 1.0 | 1.0 |
| 10 | 1.0 | 0.0 | 0.0 | 0.0 |
| 11 | 1.0 | 0.0 | 1.0 | 1.0 |
| 12 | 1.0 | 0.0 | 0.0 | 0.0 |
| 13 | 1.0 | 0.0 | 0.0 | 0.0 |
| 14 | 1.0 | 0.0 | 0.0 | 0.0 |
| 15 | 1.0 | 0.0 | 1.0 | 1.0 |
| 16 | 1.0 | 0.0 | 1.0 | 0.0 |
| 17 | 1.0 | 0.0 | 1.0 | 1.0 |
| 18 | 1.0 | 0.0 | 1.0 | 1.0 |
| 19 | 1.0 | 0.0 | 0.0 | 0.0 |
| 20 | 1.0 | 0.0 | 1.0 | 1.0 |
| 21 | 1.0 | 0.0 | 0.0 | 0.0 |
| 22 | 1.0 | 0.0 | 1.0 | 1.0 |
| 23 | 1.0 | 1.0 | 0.0 | 0.0 |
| 24 | 1.0 | 1.0 | 0.0 | 0.0 |
| 25 | 1.0 | 1.0 | 0.0 | 0.0 |
| 26 | 1.0 | 1.0 | 1.0 | 0.0 |
| 27 | 1.0 | 1.0 | 1.0 | 0.0 |
| 28 | 1.0 | 1.0 | 1.0 | 0.0 |

Table C.6: Integration Process Data

| Proj. ID | Integ Stgy | Rgrss Stgy | Integ Tests | Integ Systm | Tst Integ | Tst Rgrss |
|---|---|---|---|---|---|---|
| 1 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| 2 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| 3 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| 4 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| 5 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| 6 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| 7 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| 8 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| 9 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| 10 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| 11 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| 12 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| 13 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| 14 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| 15 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| 16 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| 17 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| 18 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| 19 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| 20 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| 21 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| 22 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| 23 | 1.0 | 1.0 | 0.0 | 1.0 | 0.0 | 1.0 |
| 24 | 1.0 | 1.0 | 0.0 | 1.0 | 0.0 | 1.0 |
| 25 | 1.0 | 1.0 | 0.0 | 1.0 | 0.0 | 1.0 |
| 26 | 1.0 | 1.0 | 0.0 | 1.0 | 0.0 | 1.0 |
| 27 | 1.0 | 1.0 | 0.0 | 1.0 | 0.0 | 1.0 |
| 28 | 1.0 | 1.0 | 0.0 | 1.0 | 0.0 | 1.0 |

Table C.7: Testing Process Data

| Proj. ID | Develop Tst Stgy | Develop Systm Tsts | Tst Systm | Regrss Tst Systm |
|---|---|---|---|---|
| 1 | 1.0 | 1.0 | 1.0 | 0.0 |
| 2 | 1.0 | 1.0 | 1.0 | 0.0 |
| 3 | 1.0 | 1.0 | 1.0 | 0.0 |
| 4 | 1.0 | 1.0 | 1.0 | 0.0 |
| 5 | 1.0 | 1.0 | 1.0 | 0.0 |
| 6 | 1.0 | 1.0 | 1.0 | 0.0 |
| 7 | 1.0 | 1.0 | 1.0 | 0.0 |
| 8 | 1.0 | 1.0 | 1.0 | 0.0 |
| 9 | 1.0 | 1.0 | 1.0 | 0.0 |
| 10 | 1.0 | 1.0 | 1.0 | 0.0 |
| 11 | 1.0 | 1.0 | 1.0 | 0.0 |
| 12 | 1.0 | 1.0 | 1.0 | 0.0 |
| 13 | 1.0 | 1.0 | 1.0 | 0.0 |
| 14 | 1.0 | 1.0 | 1.0 | 0.0 |
| 15 | 1.0 | 1.0 | 1.0 | 0.0 |
| 16 | 1.0 | 1.0 | 1.0 | 0.0 |
| 17 | 1.0 | 1.0 | 1.0 | 0.0 |
| 18 | 1.0 | 1.0 | 1.0 | 0.0 |
| 19 | 1.0 | 1.0 | 1.0 | 0.0 |
| 20 | 1.0 | 1.0 | 1.0 | 0.0 |
| 21 | 1.0 | 1.0 | 1.0 | 0.0 |
| 22 | 1.0 | 1.0 | 1.0 | 0.0 |
| 23 | 1.0 | 0.0 | 1.0 | 1.0 |
| 24 | 1.0 | 0.0 | 1.0 | 1.0 |
| 25 | 1.0 | 0.0 | 1.0 | 1.0 |
| 26 | 1.0 | 0.0 | 1.0 | 1.0 |
| 27 | 1.0 | 0.0 | 1.0 | 1.0 |
| 28 | 1.0 | 0.0 | 1.0 | 1.0 |

Table C.8: Design Complexity Data

| Proj. ID | DIT | Num Compnts | Num Objs | Num Ext I/F | I/F Prots | I/F Fmts | Num Spec Changes | Num Dsgn Changes |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 9 | 22 | 1 | 1 | 1 | 0 | 0 |
| 2 | 0 | 1 | 4 | 0 | 0 | 0 | 4 | 2 |
| 3 | 0 | 8 | 8 | 1 | 1 | 1 | 0 | 2 |
| 4 | 1 | 3 | 3 | 0 | 0 | 0 | 2 | 2 |
| 5 | 1 | 5 | 5 | 1 | 1 | 1 | 0 | 1 |
| 6 | 0 | 20 | 22 | 3 | 1 | 1 | 9 | 42 |
| 7 | 1 | 4 | 4 | 0 | 0 | 0 | 2 | 0 |
| 8 | 3 | 6 | 12 | 0 | 0 | 0 | 4 | 8 |
| 9 | 1 | 4 | 4 | 0 | 0 | 0 | 1 | 0 |
| 10 | 2 | 2 | 2 | 0 | 0 | 0 | 2 | 0 |
| 11 | 1 | 6 | 6 | 2 | 1 | 1 | 0 | 0 |
| 12 | 1 | 17 | 17 | 2 | 0 | 3 | 0 | 0 |
| 13 | 1 | 2 | 2 | 0 | 1 | 1 | 2 | 2 |
| 14 | 3 | 8 | 33 | 1 | 1 | 2 | 0 | 0 |
| 15 | 2 | 7 | 14 | 1 | 1 | 1 | 1 | 12 |
| 16 | 1 | 7 | 8 | 6 | 0 | 4 | 7 | 8 |
| 17 | 1 | 3 | 16 | 1 | 1 | 1 | 40 | 1 |
| 18 | 3 | 14 | 28 | 2 | 1 | 0 | 0 | 3 |
| 19 | 3 | 3 | 9 | 3 | 2 | 5 | 5 | 1 |
| 20 | 3 | 3 | 3 | 0 | 0 | 0 | 0 | 0 |
| 21 | 0 | 3 | 2 | 1 | 0 | 0 | 21 | 1 |
| 22 | 0 | 12 | 11 | 1 | 1 | 6 | 19 | 1 |
| 23 | 2 | 3 | 26 | 1 | 1 | 3 | 3 | 0 |
| 24 | 2 | 3 | 39 | 1 | 1 | 3 | 0 | 0 |
| 25 | 3 | 4 | 47 | 1 | 1 | 11 | 0 | 0 |
| 26 | 3 | 4 | 60 | 1 | 1 | 1 | 0 | 0 |
| 27 | 3 | 5 | 68 | 1 | 1 | 1 | 0 | 0 |
| 28 | 3 | 6 | 77 | 1 | 1 | 1 | 1 | 0 |

Table C.9: Suitability Data

| Proj. ID | Num Specs | Num Specs Intend Vrfy | Num Specs Vrfy | Num Faults | Num Missing Specs | Num Incorr Imp Specs |
|---|---|---|---|---|---|---|
| 1 | 54 | 54 | 54 | 0 | 0 | 0 |
| 2 | 13 | 12 | 12 | 0 | 1 | 0 |
| 3 | 63 | 63 | 63 | 0 | 0 | 0 |
| 4 | 30 | 21 | 21 | 1 | 9 | 0 |
| 5 | 42 | 30 | 30 | 0 | 12 | 0 |
| 6 | 30 | 23 | 23 | 0 | 7 | 0 |
| 7 | 17 | 9 | 9 | 2 | 8 | 0 |
| 8 | 35 | 28 | 28 | 1 | 7 | 0 |
| 9 | 23 | 23 | 20 | 2 | 0 | 3 |
| 10 | 23 | 23 | 23 | 0 | 0 | 0 |
| 11 | 13 | 13 | 11 | 1 | 0 | 2 |
| 12 | 38 | 19 | 19 | 4 | 19 | 0 |
| 13 | 32 | 22 | 21 | 1 | 10 | 1 |
| 14 | 28 | 13 | 13 | 1 | 15 | 0 |
| 15 | 38 | 30 | 30 | 1 | 8 | 0 |
| 16 | 63 | 63 | 63 | 1 | 0 | 0 |
| 17 | 40 | 28 | 28 | 1 | 12 | 0 |
| 18 | 26 | 22 | 14 | 1 | 4 | 8 |
| 19 | 15 | 6 | 6 | 1 | 9 | 0 |
| 20 | 10 | 0 | 0 | 1 | 10 | 0 |
| 21 | 23 | 22 | 22 | 1 | 1 | 0 |
| 22 | 33 | 30 | 30 | 1 | 3 | 0 |
| 23 | 25 | 24 | 24 | 1 | 1 | 0 |
| 24 | 9 | 9 | 9 | 1 | 0 | 0 |
| 25 | 19 | 18 | 18 | 1 | 1 | 0 |
| 26 | 9 | 9 | 9 | 1 | 0 | 0 |
| 27 | 21 | 18 | 18 | 1 | 3 | 0 |
| 28 | 19 | 19 | 19 | 1 | 0 | 0 |

Table C.10: Accuracy and Interoperability Data

| Proj. ID | Accry Specs | Accry Vrfy | Precn Specs | Precn Vrfy | DatXg Specs | DatXg Vrfy | IFPrt Specs | IFPrt Vrfy |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 2 | 2 | 1 | 2 | 2 | 0 | 0 |
| 2 | 2 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| 5 | 2 | 3 | 2 | 2 | 1 | 1 | 1 | 1 |
| 6 | 2 | 0 | 0 | 23 | 0 | 0 | 0 | 0 |
| 7 | 4 | 4 | 4 | 4 | 0 | 0 | 0 | 0 |
| 8 | 0 | 28 | 2 | 28 | 1 | 6 | 0 | 0 |
| 9 | 8 | 7 | 8 | 7 | 0 | 7 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 | 5 | 6 | 0 | 0 |
| 11 | 0 | 1 | 0 | 1 | 0 | 2 | 4 | 2 |
| 12 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 |
| 13 | 0 | 0 | 0 | 0 | 10 | 0 | 0 | 0 |
| 14 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| 15 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 17 | 0 | 7 | 0 | 7 | 0 | 7 | 0 | 7 |
| 18 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 19 | 1 | 2 | 1 | 2 | 1 | 5 | 1 | 2 |
| 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 21 | 5 | 1 | 2 | 0 | 4 | 0 | 0 | 0 |
| 22 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 23 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 24 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 25 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 26 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 27 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 28 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table C.11: Security and Compliance Data

| Proj. ID | AcAud Specs | AcAud Vrfy | AcCtl Specs | AcCtl Vrfy | Ecypt Specs | Ecypt Vrfy | FComp Specs | FComp Vrfy |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 4 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 1 | 0 | 2 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 9 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 11 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 12 | 2 | 2 | 2 | 1 | 0 | 1 | 0 | 0 |
| 13 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 5 |
| 14 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 15 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 17 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6 |
| 18 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 19 | 0 | 3 | 0 | 2 | 0 | 4 | 1 | 3 |
| 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 21 | 1 | 1 | 1 | 1 | 0 | 0 | 3 | 1 |
| 22 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 23 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 24 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 25 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 26 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 27 | 0 | 0 | 3 | 3 | 0 | 0 | 0 | 0 |
| 28 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table C.12: Reliability Data

| Proj.<br>ID | FtAvd<br>Specs | FtAvd<br>Vrfy | OpAvd<br>Specs | OpAvd<br>Vrfy | Restr<br>Specs | Restr<br>Vrfy | RComp<br>Specs | RComp<br>Vrfy |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| 5 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 6 | 0 | 16 | 0 | 16 | 0 | 0 | 0 | 0 |
| 7 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 1 | 6 | 0 | 1 | 0 | 2 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10 | 0 | 2 | 1 | 2 | 1 | 2 | 0 | 0 |
| 11 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 12 | 0 | 6 | 0 | 0 | 0 | 0 | 0 | 0 |
| 13 | 0 | 2 | 0 | 8 | 0 | 0 | 0 | 2 |
| 14 | 0 | 0 | 1 | 18 | 1 | 0 | 0 | 18 |
| 15 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 16 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 17 | 0 | 2 | 0 | 2 | 0 | 2 | 0 | 2 |
| 18 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 19 | 1 | 5 | 0 | 5 | 0 | 4 | 2 | 5 |
| 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 21 | 2 | 1 | 2 | 1 | 2 | 1 | 1 | 1 |
| 22 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 23 | 0 | 0 | 0 | 0 | 2 | 2 | 0 | 0 |
| 24 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 25 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 26 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 27 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 28 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

Table C.13: Efficiency Data

| Proj.<br>ID | TmBvr<br>Specs | TmBvr<br>Vrfy | IOUtl<br>Specs | IOUtl<br>Vrfy | EComp<br>Specs | EComp<br>Vrfy |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 |
| 11 | 0 | 0 | 0 | 0 | 0 | 0 |
| 12 | 0 | 0 | 0 | 0 | 0 | 0 |
| 13 | 2 | 2 | 0 | 0 | 4 | 0 |
| 14 | 0 | 0 | 1 | 2 | 0 | 0 |
| 15 | 0 | 0 | 0 | 0 | 0 | 0 |
| 16 | 0 | 0 | 0 | 5 | 0 | 0 |
| 17 | 0 | 0 | 0 | 0 | 0 | 1 |
| 18 | 0 | 0 | 0 | 0 | 0 | 0 |
| 19 | 0 | 1 | 1 | 0 | 1 | 2 |
| 20 | 0 | 0 | 0 | 0 | 0 | 0 |
| 21 | 1 | 1 | 0 | 0 | 1 | 1 |
| 22 | 3 | 0 | 0 | 0 | 0 | 0 |
| 23 | 1 | 1 | 0 | 0 | 0 | 0 |
| 24 | 0 | 0 | 0 | 0 | 0 | 0 |
| 25 | 0 | 0 | 0 | 0 | 0 | 0 |
| 26 | 1 | 1 | 0 | 0 | 0 | 0 |
| 27 | 0 | 0 | 0 | 0 | 0 | 0 |
| 28 | 0 | 0 | 0 | 0 | 0 | 0 |

Table C.14: Usability Data

| Proj.<br>ID | UsCnc<br>Specs | UsCnc<br>Vrfy | SyMon<br>Specs | SyMon<br>Vrfy | UComp<br>Specs | UComp<br>Vrfy |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 5 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 1 | 0 | 0 | 0 |
| 6 | 0 | 16 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 3 | 4 | 0 | 1 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10 | 1 | 0 | 0 | 0 | 0 | 0 |
| 11 | 0 | 0 | 0 | 0 | 0 | 0 |
| 12 | 0 | 4 | 0 | 0 | 0 | 0 |
| 13 | 0 | 0 | 0 | 0 | 0 | 2 |
| 14 | 0 | 0 | 0 | 4 | 1 | 0 |
| 15 | 0 | 0 | 0 | 0 | 0 | 0 |
| 16 | 0 | 1 | 0 | 0 | 0 | 0 |
| 17 | 0 | 6 | 0 | 0 | 0 | 7 |
| 18 | 0 | 0 | 0 | 0 | 0 | 0 |
| 19 | 1 | 1 | 0 | 1 | 1 | 1 |
| 20 | 0 | 0 | 0 | 0 | 6 | 0 |
| 21 | 0 | 1 | 0 | 0 | 0 | 1 |
| 22 | 0 | 0 | 0 | 0 | 0 | 0 |
| 23 | 0 | 0 | 0 | 0 | 2 | 2 |
| 24 | 0 | 0 | 0 | 0 | 4 | 4 |
| 25 | 0 | 0 | 1 | 1 | 3 | 3 |
| 26 | 0 | 0 | 0 | 0 | 0 | 0 |
| 27 | 0 | 0 | 1 | 1 | 1 | 1 |
| 28 | 0 | 0 | 1 | 1 | 0 | 0 |

Table C.15: Maintainability and Portability Data

| Proj. ID | AcRec Specs | AcRec Vrfy | SWOp Specs | SWOp Vrfy | HWOp Specs | HWOp Vrfy |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 1 | 1 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 3 | 3 | 1 | 1 |
| 5 | 1 | 1 | 1 | 0 | 1 | 0 |
| 6 | 2 | 0 | 1 | 0 | 1 | 0 |
| 7 | 0 | 2 | 3 | 2 | 0 | 0 |
| 8 | 2 | 1 | 0 | 0 | 2 | 0 |
| 9 | 0 | 1 | 0 | 0 | 0 | 0 |
| 10 | 1 | 2 | 1 | 1 | 1 | 1 |
| 11 | 2 | 2 | 1 | 1 | 1 | 1 |
| 12 | 0 | 0 | 0 | 2 | 0 | 2 |
| 13 | 0 | 0 | 1 | 0 | 2 | 0 |
| 14 | 0 | 0 | 0 | 1 | 0 | 1 |
| 15 | 0 | 0 | 1 | 1 | 0 | 1 |
| 16 | 1 | 1 | 1 | 0 | 0 | 1 |
| 17 | 1 | 0 | 1 | 1 | 0 | 0 |
| 18 | 0 | 0 | 2 | 1 | 0 | 0 |
| 19 | 0 | 5 | 0 | 4 | 0 | 1 |
| 20 | 0 | 0 | 0 | 0 | 0 | 0 |
| 21 | 1 | 1 | 0 | 0 | 0 | 1 |
| 22 | 0 | 0 | 2 | 0 | 0 | 0 |
| 23 | 0 | 0 | 0 | 0 | 0 | 0 |
| 24 | 0 | 0 | 0 | 0 | 0 | 0 |
| 25 | 1 | 1 | 0 | 0 | 0 | 0 |
| 26 | 0 | 0 | 0 | 0 | 0 | 0 |
| 27 | 3 | 3 | 0 | 0 | 0 | 0 |
| 28 | 2 | 2 | 0 | 0 | 0 | 0 |

# LIST OF REFERENCES

[AFT03] P. Antal, G. Fannes, D. Timmerman, Y.Moreau, and B.DeMoor. "Bayesian applications of belief networks and multilayer perceptrons for overian tumor classification with rejection." *Artificial Intelligence in Medicine*, **29**(2):39–60, September-October 2003.

[APB02] D. Azar, D. Precup, S. Bouktif, B. Kegl, and H. Sahraoui. "Combining and adapting software quality predictive models by genetic algorithms." In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, pp. 285–288, Los Angeles, CA, September 2002. IEEE Computer Society Press.

[Bac95] J. Bach. "Enough About Process: What We Need Are Heroes." *Computer*, **12**(2):96–98, March 1995.

[Bac99] J. Bach. "What Software Reality Is Really About." *Computer*, **32**(12):148–149, December 1999.

[Bay63] T. Bayes. "Essay Towards Solving a Problem in the Doctrine of Chances." *Philisophical Transactions of the Royal Society of London*, **53**:370–418, 1763.

[BBL76] B.W. Boehm, J.R. Brown, and M. Lipow. "Quantitative Evaluation of Software Quality." In *Proceedings of the 2nd International Conference on Software Engineering*, pp. 592–605, Los Alamitos, CA, 1976. IEEE Computer Society Press.

[BBM96] V.R. Basili, L.C. Briand, and W.L. Melo. "A Validation of Object-Oriented Design Metrics as Quality Indicators." *IEEE Transactions on Software Engineering*, **22**(10):751–760, October 1996.

[BDP98] L.C. Briand, J. Daly, V. Porter, and J. Wust. "Predicting fault-prone classes with design measures in object-oriented systems." In *Proceedings of the 9th International Symposium on Software Reliability Engineering*, pp. 334–343, Paderborn, November 1998.

[Ben95] G. Ben-Yaacov. "Reap the rewards of quality with ISO 9000." *IEEE Computer Applications in Power*, **8**(4):26–30, October 1995.

[BL02] J.M. Beaver and D.G. Linton. "Using Design Metrics to Predict Error-Prone Modules." In *Proceedings of the 6th IASTED International Conference on Software Engineering and Applications*, Cambridge, MA, November 2002. ACTA Press.

[BMB94] L. Briand, S. Morasca, and V.R. Basili. “Defining and Validating High-Level Design Metrics.” Technical report, University of Maryland, USA, 1994.

[Boe81] B. Boehm. *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[BS97] A. Berler and S.E. Shimony. “Bayes Networks for Sonar Sensor Fusion.” In *Proceedings of the 13th Conference on Uncertainty in Artificial Intelligence*, pp. 14–21. Morgan Kaufmann, August 1997.

[BS03a] J.M. Beaver and G.A. Schiavone. “A Comparison of Software Quality Modeling Techniques.” In *Proceedings of the International Conference on Software Engineering Research and Practice*, Las Vegas, NV, June 2003. CSREA Press.

[BS03b] J.M. Beaver and G.A. Schiavone. “Spatial Data Analysis as a Software Quality Modeling Technique.” In *Proceedings of the Fifteenth International Conference on Software Engineering and Knowledge Engineering*, San Francisco, CA, July 2003. Knowledge Systems Institute.

[BS06] J.M. Beaver and G.A. Schiavone. “The Effects of Development Team Skill on Software Product Quality.” *ACM SIGSOFT Softw. Eng. Notes*, **31**(3):1–5, May 2006.

[BSB05] J.M. Beaver, G.A. Schiavone, and J.S. Berrios. “Predicting Software Suitability Using a Bayesian Belief Network.” In *4th International Conference on Machine Learning and Applications*, Los Angeles, CA, December 2005. IEEE Computer Society Press.

[BSC89] I.A. Beinlich, H.J. Suermondt, R.M. Chavez, and G.F. Cooper. “The ALARM Monitoring System: A Case Study with Two Probabilistic Inference Techniques for Belief Networks.” In *Proceedings of the 2nd European Conference on Artificial Intelligence in Medicine*, pp. 247–256. Springer-Verlag, 1989.

[BVT03] R.K. Bandi, V.K. Vaishanavi, and D.E. Turk. “Predicting Maintenance Performance Using Object-Oriented Design Complexity Metrics.” *IEEE Transactions on Software Engineering*, **29**(1):77–87, January 2003.

[CC00a] B. Cukic and D. Chakravarthy. “Bayesian framework for reliability assurance of a deployed safety critical system.” In *Proceedings of the 5th International IEEE Symposium on High Assurance Systems Engineering*, pp. 321–329, Los Angeles, CA, November 2000. IEEE Computer Society Press.

[CC00b] B. Cukic and D. Chakravarthy. “Bayesian framework for reliability assurance of a deployed safety critical system.” In *Proceedings of the 5th International IEEE Symposium on High Assurance Systems Engineering*, pp. 321–329, Los Angeles, CA, November 2000. IEEE Computer Society Press.

[CDS86] S.D. Conte, H.E. Dunsmore, and V.Y. Shen. *Software Engineering Metrics and Models*. Benjamin/Cummings, Menlo Park, CA, 1986.

[CG93] R.E. Courtney and D.A. Gustafson. "Shotgun correlations in software measures." *Softw. Eng. J.*, **8**(1):5–13, January 1993.

[Chu01] S. Chulani. "Bayesian analysis of software cost and quality models." In *Proceedings of the IEEE International Conference on Software Maintenance, 2001*, pp. 565–568, Los Angeles, CA, November 2001. IEEE Computer Society Press.

[CK94] S. R. Chidamber and C.F. Kemerer. "A Metrics Suite for Object-Oriented Design." *IEEE Transactions on Software Engineering*, **20**(6):476–493, June 1994.

[Cor06] Norsys Software Corporation. "Netica (online)." In *http://www.norsys.com*, Vancouver, Canada, 2006. Norsys Software Corporation.

[Dol00] J.J. Dolado. "A validation of the component-based method for software size estimation." *IEEE Transactions on Software Engineering*, **26**(10):1006–1021, October 2000.

[Dro95] R.G. Dromey. "A Model for Software Product Quality." *IEEE Transactions on Software Engineering*, **21**(2):146–162, February 1995.

[Dro96] R.G. Dromey. "Cornering the Chimera [software quality]." *IEEE Software*, **13**(1):33–43, January 1996.

[DS97] M. Diaz and J. Sligo. "How software process improvement helped Motorola." *IEEE Software*, **14**(5):75–81, Sept.-Oct. 1997.

[DV99] A.M. Dean and D.T. Voss. *Design and Analysis of Experiments*. Springer-Verlag New York, Inc., New York, NY, 1999.

[EB99] K. El-Emam and L. Briand. "Costs and Benefits of Software Process Improvement." Technical report, 1999.

[EBG01] K. El Emam, S. Benlarbi, N. Goal, and S.N. Rai. "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics." *IEEE Transactions on Software Engineering*, **27**(7):630–650, July 2001.

[EKC98] Matthew Evett, Taghi Khoshgoftar, Pei der Chien, and Edward Allen. "GP-based software quality prediction." In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pp. 60–65, University of Wisconsin, Madison, Wisconsin, USA, 22-25 1998. Morgan Kaufmann.

[ES95] K.J. Ezawa and T. Schuermann. "Fraud/Uncollectible Debt Detection Using a Bayesian Network Based Learning System: A Rare Binary Outcome with Mixed Data Structures." In *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence*, pp. 157–166. Morgan Kaufmann, August 1995.

[ESL97] F. Engelmann, H. Steinan, and E. Lebsanft. "Bootstrap: Four Years of Assessment Experience." In *Proceedings of the 19th International Conference on Software Engineering*, pp. 568–569, New York, NY, May 1997. IEEE Computer Society Press.

[FKN02] N. Fenton, P. Krause, and M. Neil. "Software Measurement: Uncertainty and Causal Modeling." *IEEE Software*, **19**(4):116–122, July/August 2002.

[FLN98] N.E. Fenton, B. Littlewood, M. Neil, L. Strigini, A. Sutcliffe, and D. Wright. "Assessing Dependability of Safety Critical Systems using Diverse Evidence." *lEE Proc. Softw. Eng.*, **145**(1):35–39, February 1998.

[FN99] N.E. Fenton and M. Neil. "A Critique of Software Defect Prediction Models." *IEEE Transactions on Software Engineering*, **25**(5):675–689, September/October 1999.

[Fou06] Eclipse Foundation. "Eclipse (online)." In *http://www.eclipse.org*, Ontario, Canada, 2006. Eclipse Foundation, Inc.

[Gar06] G.D. Garson. "Correlation (online)." In *http://www2.chass.ncsu.edu/garson/PA765/correl.htm*, Raleigh, North Carolina, 2006. Cruise Scientific.

[GBB90] N. Gorla, A.C. Benander, and B.A. Benander. "Debugging effort estimation using software metrics." *IEEE Transactions on Software Engineering*, **16**(2):223–231, February 1990.

[Hal77] M.H. Halstead. *Elements of Software Science.* El Sevier North Holland, Inc., 1977.

[Hal96] T.J. Haley. "Software process improvement at Raytheon." *IEEE Software*, **13**(6):33–41, November 1996.

[Hen96] B. Henderson-Sellers. "The mathematical validity of software metrics." *SIGSOFT Softw. Eng. Notes*, **21**(5):89–94, 1996.

[HG96] J.D. Herbsleb and D.R. Goldenson. "A systematic survey of CMM experience and results." In *Proceedings of the 18th International Conference on Software Engineering*, pp. 323–330. IEEE Computer Society Press, March 1996.

[HH98] D. Heckerman and E. Horvitz. "Inferring Informational Goals from Free-Text Queries: A Bayesian Approach." In *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence*, pp. 230–237. Morgan Kaufmann, July 1998.

[HHH92] D.E. Heckerman, E.J. Horvitz, and B.N. Hathwani. "Toward Normative Expert Systems: Part I: The Pathfinder Project." *Methods of Information in Medicine*, **31**:90–105, 1992.

[HJP98] A.M.J. Hass, J. Johansen, and J. Preis-Heje. "Does ISO 9001 increase software development maturity?" In *Proceedings of the 24th Euromicro Conference, 1998*, pp. 860–866, August 1998.

[HK81] S. Henry and D. Kafura. "Software Structure Metrics Based on Information Flow." *IEEE Transactions on Software Engineering*, **SE-7**(5):510–517, September 1981.

[HK84] S. Henry and D. Kafura. "The Evaluation of Software Systems' Structure Using Quantitative Software Metrics." *Softw. - Practice and Experience*, **14**(6):561–573, June 1984.

[HMK94] V. Haase, R. Messnarz, G. Koch, H.J. Kugler, and P. Decrinis. "Bootstrap: finetuning process assessment." *IEEE Software*, **11**(4):25–35, July 1994.

[HS87] W.S. Humphrey and W.L. Sweet. "A Method for Assessing the Software Engineering Capability of Contractors." Technical report, Software Engineering Institute, Pittsburgh, PA, 1987.

[Ins02] Software Engineering Institute. *Capability Maturity Model Integrated, Version 1.1.* Carnegie Mellon University, Pittsburgh, PA, 2002.

[ISO98] ISO/IEC 15504. *ISO/IEC 15504:1998. Information Technology - Software Process Assessment.* International Organization for Standardization, Geneva, Switzerland, 1998.

[ISO01] ISO/IEC 9126. *ISO/IEC 9126:2001. Software Engineering - Product Quality.* International Organization for Standardization, Geneva, Switzerland, 2001.

[Jen96] F.V. Jensen. *An Introduction to Bayesian Networks.* UCL Press Limited, 1996.

[Jon94] C. Jones. "Software metrics: good, bad, and missing." *Computer*, **27**(9):98–100, September 1994.

[Jon96] C. Jones. "The economics of software process improvement." *Computer*, **29**(1):95–97, January 1996.

[KAB96] T.M. Khoshgoftaar, E.B. Allen, L.A. Bullard, R. Halstead, and G.P. Trio. "A tree-based classification model for analysis of a military software system." In *Proceedings of the High-Assurance Systems Engineering Workshop, 1996*, pp. 244–251. IEEE Computer Society, October 1996.

[KAH97] T.M. Khoshgoftaar, E.B. Allen, J.P. Hudepohl, and S.J. Aud. "Application of Neural Networks to Software Quality Modeling of a Very Large Telecommunications System." *IEEE Trans. Neural Networks*, **8**(4):902–909, July 1997.

[Kan95] S.H. Kan. *Metrics and Models in Software Quality Engineering.* Addison-Wesley, 1995.

[KBR92] T.M. Khoshgoftaar, B.B. Bhattacharya, and G.D. Richardson. "Predicting Software Errors, During Development, Using Nonlinear Regression Models: A Comparative Study." *IEEE Trans. Rel.*, **41**(3):390–395, September 1992.

[KC85] D. Kafura and J. Canning. "A validation of software metrics using many metrics and two resources." In *Proceedings of the 8th International Conference on Software Engineering*, pp. 378–385. IEEE Computer Society Press, August 1985.

[KM90] T.M. Khoshgoftaar and J.C. Munson. "The line of code metric as a predictor of program faults: a critical analysis." In *Proceedings of the 14th Annual International Computer Software and Applications Conference (COMPSAC 90)*, pp. 408–413. IEEE Computer Society, November 1990.

[KMB92] T.M. Khoshgoftaar, J.C. Munson, B.B. Bhattacharya, and G.D. Richardson. "Predictive Modeling Techniques of Software Quality from Software Measures." *IEEE Transactions on Software Engineering*, **18**(5):979–987, November 1992.

[KP96] B. Kitchenham and S.L. Pfleeger. "Software Quality: The Elusive Target." *IEEE Software*, **13**(1):12–21, January 1996.

[KPL90] B.A. Kitchenham, L.M. Pickard, and S.J. Linkman. "An evaluation of some design metrics." *Softw. Eng. J.*, **5**(1):50–58, January 1990.

[KS96] T.M. Khoshgoftaar and R.M. Szabo. "Using Neural Networks to Predict Software Faults During Testing." *IEEE Trans. Rel.*, **45**(3):456–462, September 1996.

[KS02a] T.M. Khoshgoftaar and N. Seliya. "Software quality classification modeling using the SPRINT decision tree algorithm." In *Proceedings of the 14th International Conference on Tools with Artificial Intelligence, 2002 (ICTAI 2002)*, pp. 365–374. IEEE Computer Society, November 2002.

[KS02b] T.M. Khoshgosftaar and N. Seliya. "Software quality classification modeling using the SPRINT decision tree algorithm." In *Proceedings of the 14th IEEE International Conference on Tools with Artificial Intelligence*, pp. 365–374, Los Angeles, CA, November 2002. IEEE Computer Society Press.

[KSG95] T.M. Khoshgoftaar, R.M. Szabo, and P.J. Guasti. "Exploring the behaviour of neural network software quality models." *Softw. Eng. J.*, **10**(3):89–96, May 1995.

[KUS04] S. Kanmani, V.R. Uthariaraj, V. Sankaranarayanan, and P. Thambidurai. "Object oriented software quality prediction using general regression neural networks." *ACM SIGSOFT Softw. Eng. Notes*, **29**(5):1–6, September 2004.

[LH93] W. Li and S. Henry. "Object-oriented metrics that predict maintainability." *Journal of Systems and Software*, **23**:111–122, 1993.

[LLV95] F. Lanubile, A. Lonigro, and G. Visaggio. "Comparing models for identifying fault-prone software components." In *Proceedings of the 7th Anuual Conference on Software Engineering and Knowledge Engineering*, pp. 12–19. Knowledge Systems Institute, June 1995.

[Mar04] P. Marrone. "Java Object-Oriented Neural Engine (JOONE)(online)." In *http://www.jooneworld.com*, 2004.

[McC76] T. McCabe. "A Complexity Measure." *IEEE Transactions on Software Engineering*, **SE-2**(4):308–320, December 1976.

[MDC03] T. Menzies, J.S. DiStefano, and M. Chapman. "Learning early lifecycle IV and V quality indicators." In *Proceedings of the 9th International Software Metrics Symposium, 2003*, pp. 88–96. IEEE Computer Society, September 2003.

[MK92] T. Mukhopadhyay and S. Kekre. "Software effort models for early estimation of process control applications." *IEEE Transactions on Software Engineering*, **18**(10):915–924, October 1992.

[MRW77] J. McCall, P. Richards, and G. Walters. "Factors in Software Quality." Technical report, Rome Air Development Center, New York, November 1977.

[MS95] W. Mendenhall and T. Sincich. *Statistics for Engineers and the Sciences.* Prentice-Hall, Upper Saddle River, NJ, 4 edition, 1995.

[Nai82] F.A. Naib. "An application of software science to the quantitative measurement of code quality." In *Proceedings of the 1982 Workshop on Software Metrics*, pp. 101–128, March 1982.

[NAS04] NASA. *NASA Procedural Requirements 7150.2: Software Engineering Requirements.* National Aeronautics and Space Administration, Goddard Space Flight Center, MD, 2004.

[NAS05] NASA. "Competency Management System (online)." In *http://ohr.gsfc.nasa.gov/cms/home.htm*, Goddard Space Flight Center, MD, April 2005. National Aeronautics and Space Administration.

[PC89] A. Puerta and C.L. Carnal. "An exploratory study on a linear model for measuring software quality." In *Proceedings of Southeastcon '89: Energy and Information Technologies in the Southeast*, pp. 1099–1102. IEEE Computer Society Press, April 1989.

[Pea86] J. Pearl. "Fusion, Propagation, and Structuring in Belief Networks." *Artificial Intelligence*, **29**:241–288, 1986.

[QT03] Tong-Seng Quah and Mie Mie Thet Thwin. "Application of Neural Networks for Software Quality Prediction Using Object-Oriented Metrics." In *Proceedings of the International Conference on Software Maintenance, 2003*, pp. 116–125. IEEE Computer Society Press, September 2003.

[Sci06] Cruise Scientific. "Visual Statistics: Chapter 15 The Point Biserial Coefficient of Correlation (online)." In *http://www.visualstatistics.net.* Cruise Scientific, 2006.

[She90] M. Sheppard. "Design metrics: an empirical analysis." *Softw. Eng. J.*, **5**(1):3–10, January 1990.

[SK03] 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*, **29**(4):297–310, April 2003.

[SM99] G.G. Schulmeyer and J.I. McManus. *Handbook of Software Quality Assurance.* Prentice-Hall, Inc., 1999.

[Smi80] C.P. Smith. "A software science analysis of program size." In *Proceedings of the 1980 ACM Annual Conference*, pp. 437–446, January 1980.

[SP88] R.W. Selby and A.A. Porter. "Learning from examples: generation and evaluation of decision trees for software resource analysis." *IEEE Transactions on Software Engineering*, **14**(12):1743–1757, December 1988.

[SSM97] C. Smidts, D. Sova, and G.K. Mandela. "An architectural model for software reliability quantification." In *Proceedings of the 8th IEEE Symposium on Software Reliability Engineering*, pp. 324–335, Los Angeles, CA, November 1997. IEEE Computer Society Press.

[SYT85] V. Shen, T. Yu, S. Thebout, and L. Paulsen. "Identifying error-prone software - An empirical study." *IEEE Transactions on Software Engineering*, **11**:317–323, April 1985.

[TS91] W.R. Torres and M. Samadzadeh. "Software reuse and information theory based metrics." In *Proceedings of the 1991 Symposium on Applied Computing*, pp. 437–446, April 1991.

[Wit90] C. Withrow. “Error density and size in Ada software.” *IEEE Software*, **7**(1):26–31, January 1990.

[Wri21] S. Wright. “Correlation and Causation.” *Journal of Agricultural Research*, **20**:557–585, 1921.

[ZZ93] W.M. Zage and D.M. Zage. “Evaluating design metrics on large-scale software.” *IEEE Software*, **10**(4):75–81, July 1993.