AN EXTENSIBLE, HIERARCHICAL ARCHITECTURE FOR ANALYSIS OF SOFTWARE QUALITY ASSURANCE

by

David Mark Rice

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

(December, 2020)

ACKNOWLEDGEMENTS

This thesis could not have been accomplished without the incredible support and advice from my advisor, Clemente Izurieta. Your patience, management skills, openness, and ideas has provided an incredible graduate school experience. Thank you for all that you do.

I also thank my parents. Your love and support over all phases of my education has not been forgotten. I would not be where I am now without your amazing support.

Finally, I thank my roommate, Richard, for listening to many of the ideas surrounding this thesis and providing valuable ideas and conversations. Your ability to listen and conceptualize ideas I was having trouble with helped direct many of the routes this thesis took. Thank you.

TABLE OF CONTENTS

TABLE OF CONTENTS–CONTINUED

TABLE OF CONTENTS–CONTINUED

TABLE OF CONTENTS–CONTINUED

LIST OF TABLES

LIST OF TABLES–CONTINUED

Table Page

LIST OF TABLES–CONTINUED

Table Page

LIST OF TABLES–CONTINUED

Table Page

LIST OF FIGURES

Figure Page

LIST OF FIGURES–CONTINUED

Figure Page

ABSTRACT

As software becomes integrated into most aspects of life, a need to assess and guarantee the quality of a software product is paramount. Poor software quality can lead to traffic accidents, failure of life-saving devices, government destabilization, and economic ruin. To assess software quality, quality researchers design quality models. A common quality model will decompose quality concepts such as "total quality", "maintainability", and "confidentiality" into a hierarchy that can eventually be linked to specific lines of code in a software system. However, a problem persists in the domain of quality modeling: quality assessment through use of quality models is not finding acceptance by industry practitioners. This thesis reviews the weaknesses of modern modeling attempts and aims to improve the processes surrounding quality assessment from the perspective of both researchers and academic practitioners.

The analysis uses the Goal/Question/Metric paradigm. Two closely related goals are presented that aim to analyze a process of generating, validating, and operationalizing quality models for the purpose of improvement with respect to cost, experimentative capability, collaborative opportunity, and acceptability. A system is designed, PIQUE, that provides functionality to generate experimental quality models. Test cases and exercises are run on the models generated by PIQUE to supply metric data used to answer the questions and goals.

The results show that–in the context of a PIQUE-generated quality model compared to a similar non-PIQUE quality model–improvement can be achieved with respect to development cost and experimentative capability. Clear improvement was not found in the context of model operationalization difficulty and output acceptability. Ultimately, partial achievement of both goals is realized. The work concludes that the current problems in the domain of quality modeling can be improved upon, and systems like PIQUE are a valuable approach toward that goal.

CHAPTER ONE

INTRODUCTION

Software development processes and design has evolved greatly from the early years of assembly, FORTRAN, and LISP. As modern applications reach monolithic sizes with millions of lines of code, hundreds of dependencies, and nuanced design paradigms, the topic of the software quality begins to emerge. Quality, as it relates to most artifacts of human creation, can quickly become a philosophical venture.

What makes a software product good? How does one measure the goodness of a product? Is there a dogmatic, objective measure of software quality or is the very nature of quality subjective?

Tackling these questions is becoming paramount in the domain of software engineering where poor software quality can put lives or businesses at risk. Examples of spectacular failures in space flight attempts with key software oversights as a primary culprit are reviewed in an assessment of the role of software in aerospace systems. The Ariane 501 launcher exploded due to errors in the internal reference system software, the Mars Climate Orbiter exploded due to mishandling of distance unit data types in the software, and the Titan IV B-32 missions ended in failure due to "an inadequate software development, testing, and quality assurance process for the Centaur upper state". [28]

An oversight in a standardized encryption library caused the Heartbleed bug, affecting 17.5% of trusted HTTPS websites. Encrypted information behind both HTTPS and private VPNs were no longer safe and the financial implications were monumental [44].

Software is quickly integrating into all aspects of life: automobiles, city infrastructure, home automation, banking, education, government, national security, and the list goes on. One can easily imagine the concerning implications of poor software quality in many of these domains. How can one go about assuring that a software product is good? What does "good" mean?

To go about defining quality and integrating its assessment into a development environment, the use of quality models is a sensible choice. An overview of quality modeling domain and history is given in chapter 2, but to introduce an intuitive notion of how some of these models work, imagine two individuals interested in adopting a dog whom are presented a Havanese[1] named Lucy for their consideration.

Individual A has in mind that an ideal dog should be small, good with children, and fluffy while individual B wants a guard dog with traits of high aggression and strength. Both individuals are given a basic model as shown in figures 1.1 and 1.2 to help them decide if they should adopt or not. Note that the model values in the bottom layer of nodes come from measures of the dog itself, but the weights of the model–shown as values in the edges–come from the subjective quality value opinions of each individual. Thus, as the model evaluates upwards, the total quality of the same dog is different depending on the individual.

1.1 Motivation

While many good attempts have been made to assess the quality of a software product using quality models, the values generated leave a sense of unease and distrust by development teams [42] [43]. The field of software quality modeling is still young, leaving the generated quality values often misrepresented, misunderstood, and

[1] A Havanese is a small, friendly family dog known for their especially soft, silky double coat.

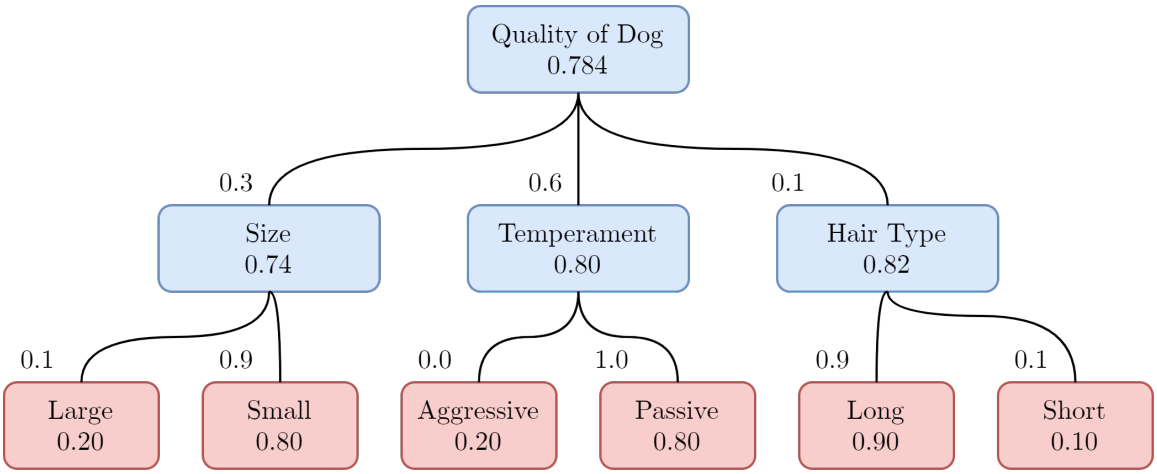

Figure 1.1: Quality assessment of Lucy the dog for individual A.

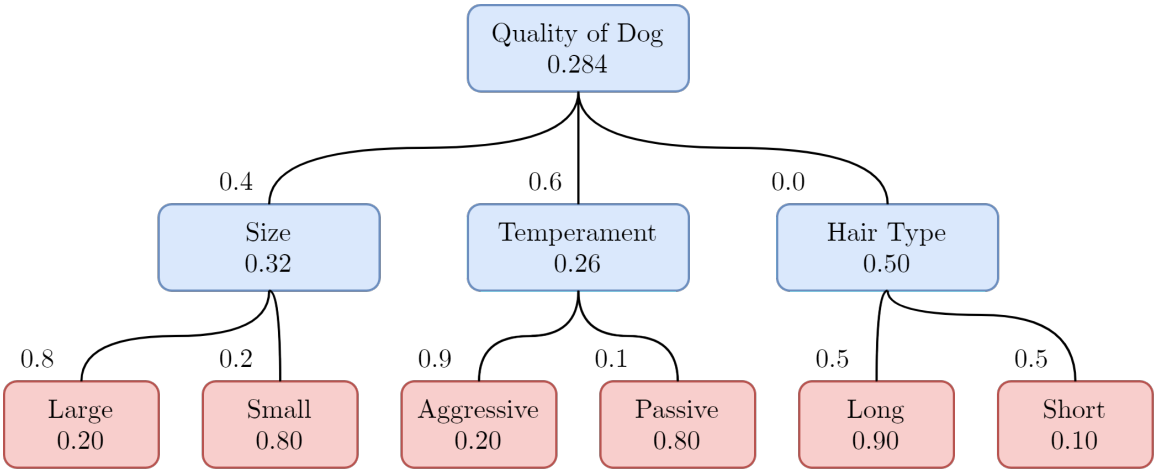

Figure 1.2: Quality assessment of Lucy the dog for individual B.

lacking academic rigor to carry meaning. Furthermore, philosophical, industrial, and academic concerns arise during use of quality modeling approaches leading to topics worth addressing.

Philosophically, this thesis argues that the definition of "good" should be made in the eye of the stakeholder, and it should accommodate frequent redefining as company priorities and values evolve over time. Dogmatic definitions of absolute quality lead to non-pragmatic and incorrect assessments.

Industrially, the results of quality model assessment should be easy to understand at all layers. It should not require an in-house quality modeling expert to use and interpret. It should not be expensive to use for time, resources, or budget. It should integrate into the development continuous integration pipeline.

Academically, the derivation of a quality model should be as automated as possible. A platform should be in place to allow researchers to spend their time evaluating if a model is effective rather than building the model itself. The domain of modern programming languages and code analysis tools[2] are continually changing, so the platform should facilitate easy addition or removal of tools, their findings, and language support. Finally, to accommodate communication and collaboration, quality models should follow a general structure and use common terms.

1.2 Motivation Formalized: GQM

To formalize the purpose of this thesis and provide a road map for the following chapters, the Goal/Question/Metric paradigm as presented by [6] is used.

Goals Two closely related goals are presented:

> **G01**: Analyze a process of generating, validating, and operationalizing quality models for the purpose of improvement with respect to effort investment, experimentation, and collaborative opportunity from the point of view of quality model researchers in the context of static software system analysis.
>
> **G02**: Analyze a process of generating and operationalizing quality models for the purpose of improvement with respect to cost investment and acceptability

[2] In software quality modeling, these tools are most often static analysis tools. They are discussed in further detail in chapter 2.2.

from the point of view of software development practitioners in the context of static software system analysis.

The definition of a quality model is evaluated in chapter 2 while key revelations about identified issues with the current state of quality models are discussed in chapter 3. The goals distinguish between the point of view of a researcher and the point of view of a development practitioner due to an important pragmatic schism: there is a difference between a good quality model and a good quality model that practitioners also trust and want to use.

Questions Regarding goal *G01*, the following questions are asked:

Q01-01: How much effort does it take to generate a model using default mechanisms?

Q01-02: How much effort does it take to generate a model using modified mechanisms?

Q01-03: How much effort does it take to operationalize a model?

Q01-04: Do the models produced facilitate ease of evaluation by researchers?

Q01-05: Are the models produced valid?

For goal *G02*, similar questions are asked; however, they take the industry's perspective:

Q02-01: Is it expensive to tune a model to a company's needs?

Q02-02: How acceptable is it to integrate quality model assessment into an external, continuous integration system?

Q02-03: Can quality models be used such that their output values are trusted by practitioners?

Metrics Due to the similar nature of both goals, a variety of metrics can apply to answering both collections of questions.[3]

Effort Metrics

M01: Man-hours taken to install PIQUE as a library resource.

M02: Man-hours taken to connect static analysis tools to PIQUE.

M03: Man-hours taken to design a quality model description *.json* file.

M04: Man-hours taken to prepare a benchmark repository.

M05: Man-hours taken to modify a default normalization function.

M06: Man-hours taken to modify a default utility function.

M07: Man-hours taken to modify a default weighting function.

M08: Man-hours taken to modify a default evaluation function.

M09: Man-hours taken to operationalize a derived quality model.

M10: Time taken to run model derivation (benchmark repo size = 44).

M11: Time taken to run system quality assessment.

M12: Time taken by a practitioner to express subjective quality definitions.

Quality Assessment Metrics

M13: The change in quality assessment score after introducing flaws.

M14: The change in quality assessment score after introducing improvements.

[3]Some terms used by these metrics are defined in future chapters.

M15: The change in quality assessment score after modifying subjective quality opinion.

Design Metrics

M16: Model output exposure.

M17: Number of external dependencies needed for actualization.

M18: [True | False] Failure of assessment does not interfere with other system processes

Subjective Metrics

M19: [True | False] The slope of assessed values over time matches practitioner opinion.

Requirement Metrics

M20: [True | False] The platform is open source.

1.3 Contributions

From a software engineering product point of view, this thesis presents a platform, PIQUE,[4] designed to facilitate a researcher's ability to quickly generate experimental quality models that are easy to operationalize. This platform serves as the catalyst to garner metrics, answer questions, and ultimately achieve the goals mentioned in section 1.2. The primary contributions are:

- Provision of a supported, open-source platform capable of generating experimental quality models using a language agnostic approach.

[4] "A Platform for Investigative software Quality Understanding and Evaluation"

- Walk-through of generation of an ISO/IEC 25010 based C# quality model from start to finish.
- Demonstration of using the platform to generate a quality model capable of adaption to subjective quality opinion.
- Demonstration of using the platform to generate a potential novel, security-focused quality model.
- Feedback from industry practitioners regarding absolute quality values compared to relative quality values.
- Validation of a C# model generated by the platform through *in vivo* and *in vitro* test cases.
- Operationalization of a model generated by the platform into a real-world system as a part of a continuous integration pipeline.
- Analysis of the platform's capabilities with respect to improvement from the context of quality model researchers.
- Analysis of the platform's capabilities with respect to improvement from the context of industry application.

This thesis is organized as follows. First, the fundamental concepts backing quality assessment and a history of quality modeling attempts are given in chapter 2. Two notable quality assessment attempts, Quamoco and QATCH, are reviewed in detail in chapter 3. These chapters are necessary, not only to build the required technical knowledge needed to understand the mechanisms of PIQUE, but also to understand the elasticity required by PIQUE's design to facilitate experimental design in quality modeling.

Next, a technical review of the design and use of PIQUE is given in chapter 4. The chapter presents a language-agnostic system that bears the load of all things related to quality model derivation and operationalization and sets the mechanisms necessary to achieve the research goals.

Chapter 5 presents a language-specific quality model instantiation of PIQUE. Specifically, this is an ISO/IEC 25010 based C# quality model with full tool support for operationalization. Test cases are run on the model to generate data to support the GQM metrics. This model is used to assess the quality of lab-designed systems for a variety of test cases which in turn provide data for the GQM metrics. A second, security-focused, quality model is also presented, but not operationalized. The security model is niche, complex, and expresses the capability of PIQUE to support experimental systems, but also shows the need for further tool support before some models can be operationalized.

Finally, the tests, analysis, and discussion of the goals, questions, and metrics presented are given in chapters 6 and 7 followed by a review of the threats to validity and final conclusions in chapters 8 and 9.

CHAPTER TWO

BACKGROUND

2.1 Quality Modeling

Quality modeling has a history reaching back to the 1970's [7], yet it remains a relatively new area of study in software engineering. Furthermore, the output from quality model assessment remains distrusted in industry [43]. Given this thesis will present a quality modeling platform whose processes will be assessed for the purpose of improvement, an understanding of previous quality modeling attempts–their successes, failures, and design decisions–becomes important.

Purpose Before expressing how quality modeling works, it is valuable to understand its common purposes and intended uses. In a publication presenting an operationalized product quality model the authors aptly say,

> "Software quality models tackle the issues [of bad software economic impact, failures of software, increased maintenance costs, high resource consumption, long test cycles, and waiting times for users] by providing a systematic approach for modeling quality requirements, analyzing and monitoring quality, and directing quality improvement measures. They thus allow ensuring quality early in the development process." [40]

In other words, the purpose of quality modeling can center around pragmatic goals such as quality improvement, quality assessment, and evaluating the financial state or implications of the system at hand. Quality improvement and assessment is driven by having a way to identify the "good" and "bad" parts of a system and monitoring the addition or removal of these parts over time. A system's financial state or implications

can refer to additional fiscal costs resulting from poor quality in the system. Common terms used with financial quality evaluation are remediation costs [27] and technical debt [9] [21] [24].

Domain Quality reveals itself in numerous domains in software engineering. Code quality, product quality-in-use, and process quality all need their own methods for quality assessment. This thesis focuses strictly on the domain of static code quality.

Quality Taxonomy in Software Engineering Assigning a hierarchial system to quality classifications has a long history of attempts as discussed later in this chapter; however, ISO/IEC JTC 1[1] [14] likely provides the best modern starting point for decomposing the high level concepts of quality. The ISO/IEC 25010 model decomposes software product quality in to eight characteristics and 31 sub-characteristics as shown in Figure 2.1.

Figure 2.1: ISO/IEC 25010. Source `https://iso25000.com`

Note that the taxonomy of Figure 2.1 remains at an abstract level of quality and does not offer concrete ways to measure its decompositions. Quality modeling often looks to extend high level taxonomies presented by standards such as ISO/IEC

[1] International Organization for Standardization and the International Electrotechnical Commission Joint Technical Committee.

and attach methods of measurement and evaluation. The quality assessment process therefore has two high-level components: the creation of a quality model and the use of that quality model for product assessment.

Quality Model Creation

While many unique approaches exist for the design of a quality model and its instantiation, a common procedure used throughout this thesis takes a top-down approach as follows:

1. Decide the top node: the quality concept under evaluation; for example, total software quality, remediation cost, or maintainability.
2. Decompose the top node into a meaningful taxonomy.
3. Once the hierarchy decomposes into concepts low enough to be measured, attach measurement tools to allow these nodes to contain numerical values from the product under evaluation.
4. Define how these leaf node values aggregate to the top node evaluation.

Using a Quality Model for Product Assessment

After a model is defined–likely with nodes, weighted edges, and tools for product measurement–numerical quality values can be obtained by running the quality model on a software product. This process involves running the tools defined in the quality model on the product to obtain measurements, using those measurements to aggregate node evaluations up the quality hierarchy tree, and ultimately calculating a numerical assessment value of the product's quality from the model's root node value.

2.2 Measurement

A frequent point of confusion regarding quality assessment is understanding how the real-word numerical values are obtained for the model to use at its lowest layer. This component of quality assessment is accomplished through measurement. Measurements come from one of two types, metrics or findings, and these measurements are obtained through the use of static analysis tools, dynamic analysis tools, or hand-entry.

Metrics and Findings

In the context of this thesis, metrics and findings are both numerical values obtained by running a static analysis tool audit on a software product, but they carry subtle differences.

Metrics Software metrics are evaluated through mathematical formulas backed by academic rigor that describe product-state truths. For example, the metric *LoC* (lines of code) evaluates to the number of lines of code of the product under assessment. The metric *coupling* provides a numeric representation of the degree of interdependence between software modules using a formula involving data parameters, control parameters, and number of modules [38]. Metrics are calculated at all levels of granularity–from single statements, to classes, design patterns [9], and complete architectures.

Findings Findings, on the other hand, represent faulty or exemplary blocks of code. Faulty lines of code can refer to code smells [30], program breaking flaws, or any other negative influence a tool can be designed to detect. The numerical evaluations of findings are often simply a count of the findings multiplied by a severity level.

For example, consider a *locked instance* finding found 10 times in the product under assessment, and *locked instance* is labeled as having a severity level of 1 (low). The value of this finding would then equal 10, likely additionally normalized by lines of code. The approach used in this thesis takes a more nuanced and flexible approach to finding evaluation functions as discussed in chapter 4.

Static Analysis Tools

Static analysis tools are external resources that audit the product under assessment to retrieve metrics or finding data by parsing byte code or compiled source code. By definition, static analysis tools do not measure dynamic aspects such as network throughput.

Some languages, such as Java, have robust static analysis tool support with tens of thousands of potential metrics or findings to detect. However, as languages become less common or more proprietary, static analysis tool capabilities drop dramatically. This can be a major weakness of quality assessment as a language-specific model can only be as strong as the tools it can utilize, and writing one's own tools or expanding the ruleset of an existing tool can be an extremely time-consuming process.

2.3 Benchmark Repositories and Utility Functions

While static analysis tools do assess the product under evaluation and return numerical values, these metrics values and findings lack meaning as individual values. Some quality modeling attempts look to bring meaning and context to the goodness or badness of these measurement values by using a collection of representative projects[2] to express the expected range of any given measure.

[2]This collection is commonly called a benchmark repository

Consider a measure, *Coupling*, which can be directly represented by a metric also named *coupling* following the formula given by [38]. The project under assessment could return a value $coupling = 0.70$. While it is known that lower values represent less coupling which is considered good, is $coupling = 0.70$ a low or high value? In this example, a benchmark repository is used by obtaining the coupling value from a large collection of other systems.[3] The benchmark repository reveals that the lowest coupling value found in its products was $coupling = 0.72$ and the highest value found was $coupling = 0.98$.

A utility function representing coupling, such as the linear utility function shown in Figure 2.2, is then generated using a min value of 0.72 and a max value of 0.98. To use the function, input the coupling value found from the product under assessment to the x-axis and return the y-axis value according to the utility function curve represented by the solid line. For the input value of $coupling = 0.70$, the utility function returns a value of 1.0 meaning the product under assessment has very good coupling.[4] If the input value was $coupling = 0.85$, the returned utility value would be 0.50, and if $coupling = 0.99$, the returned utility value would be 0.0, implying the coupling is as bad as it can be with respect to the values found in the benchmark repository.

2.4 Quality Control Loop

For a quality model to achieve its purpose in practice, implementation of a quality control loop should be developed. The quality control loop and quality improvement paradigm presented by [11] and [12] state that continuous quality

[3] The benchmark repository size generally ranges between 25 to 100 products, but needs to contain at least 1.3 million lines of code according to a study by [37].

[4] In the coupling formula given by [38] the lowest possible coupling–a module with a single input and output–is 0.67.

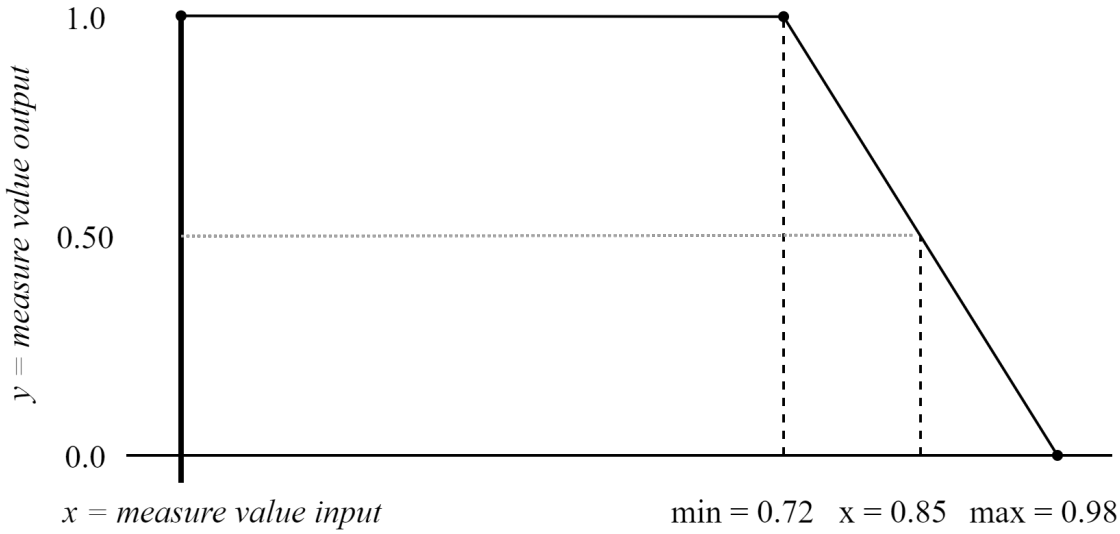

Figure 2.2: Linearly decreasing utility function

assessment during the development phase can assist in detecting and fixing quality problems early. The earlier a problem is detected the less expensive it is to fix. Additionally, consistent quality feedback can help developers recognize their good and bad implementation habits. In order to accomplish this, as many components as possible should be automated, and assessment results need to be easily understood.

Continuous Integration

Continuous integration is a development cycle involving code integration with a central repository. In general cases, new code goes through an automated pipeline of test suites, quality validation, and automated deployment. Modern solutions for continuous integration and deployment are becoming widely adopted in industry with tools such as Microsoft Azure DevOps[5], GitLab CI/CD[6], and Jenkins[7].

Using continuous integration as the technological driver for the quality control

[5] `https://azure.microsoft.com/`
[6] `https://gitlab.com/`
[7] `https://jenkins.io/`

loop is a clear solution. This will fulfill the pragmatic element of quality modeling by providing awareness in changes in quality as the changes happen and driving action items for the maintenance of quality priorities.

2.5 History of Quality Modeling

A quality model may or may not be a hierarchial decomposition of characteristics of quality. This section reviews historical and notable quality models covering a wide range of conceptual designs and paradigms. Two recent quality model approaches, Quamoco and QATCH, are the primary influences for the work of this thesis and are described in detail in chapter 3.

Early Contributions

Boehm [7] and Grady [16] present some of the earliest quality model approaches dating back to the 1970s-1980s. Both present a hierarchial approach using terms similar to what is seen in modern ISO/IEC decompositions such as maintainability, reliability, functionality, usability, and supportability. These foundational models suggest that hierarchial decomposition is an effective and intuitive way to construct quality models; however, these early attempts do not have enough low-level measurement support to produce meaningful quality values.

Dromey [13] presents a quality model that uses a hierarchy based on structural concepts of a system (class, function, object) instead of quality concepts (maintainability, reliability, usability). Of particular interest is his direct linking of a metric to a high-level quality concept by measuring the maintainability of a structural concept by a maintainability index metric. This begs the question if direct measurement of a high-level quality concept violates the construct validity of a quality measurement. If not, then quality models can be constructed with very few layers with their metrics

directly representing their quality.

Early ISO/IEC 9126 and 25010 Based Extensions

The software quality taxonomy provided by ISO/IEC from 9126 and onward [14] became the accepted structural starting point for most quality modeling attempts. Of particular focus are eight subcharacteristics.

- Functional Suitability
- Performance Efficiency
- Compatibility
- Usability
- Reliability
- Security
- Maintainability
- Portability

The ISO/IEC taxonomy presented represents only the top half of what is needed to measure quality, thus it is the job of quality modelers to extend the model into parts that can be meaningfully measured.

In 1996, Van Zeist [39] presents an extension of the ISO 9126 model using a measurable concept called indicators. A useful measure, average learning time, is also provided. In 2003, Franch [15] presents a variety of interesting metrics in an attempt to use quality models to assess the quality of software package selection. Samoladas [35] presents a useful extension of ISO 9126 in 2008 specifically focused on open source software evaluation. This paper presents the transformation of numerical

quality values into an ordinal scale representation, a method many modern approaches use as well.

While these models continue to expand the field of research in quality modeling, a critique lies in their use of metrics for low-level measurements. In light of the granularity and scope these quality models need to represent, the representation and strength metrics of their associated low-level model nodes often leaves much to be doubted.

Modern Notable Contributions

The SQUID project [25] from 1997 is an ISO/IEC 9126 based approach to quality model definitions. The project takes a holistic approach to quality modeling, showing the need for a meta model to guide quality model construction. Additionally, it presents a componentized quality model specifically designed to customize to individual project needs.

The work of Bansiya and Davis [4] in 2002 continues the early 1995 work of Dromey [13], but presents a quality model specifically focused on object oriented systems. The work also presents a significant addition of metrics and tool support.

The SIG maintainability model [17] in 2007 introduces a model that does not use the full ISO/IEC quality taxonomy at its highest level. Instead, only a maintainability model is presented. Subcharacteristics for maintainability are compared against five source code properties: volume, complexity per unit, duplication, unit size, and unit testing.

The SQUALE model and framework [31] from 2009 is an ISO/IEC 9126 based quality model with notable focus put on usefulness to practitioners in industry. SQUALE extends ISO/IEC 9126 through a more granular middle layer using a concept called practices. Along with providing further metrics for low level evaluation,

the framework offers useful tool support and visualization options. The quality model used, however, does not modularize and does not offer a strong connection from low quality scores to the low-level code issues responsible.

The similarly named SQALE[8] model [27] from 2010 takes a different approach to quality modeling by linking quality to technical debt values based on remediation costs. The model is structured in ranked layers using calculated values of testability, reliability, changeability, efficiency, maintainability, and reusability. A modern popular framework and continuous integration quality monitoring service, SonarQube,[9] uses SQALE as its underlying quality model for assessment.

Potential Uses for Quality Modeling

Given that the concept of quality is a remarkably abstract concept that can apply to nearly anything, the use of modeling quality concepts can go far beyond the ISO/IEC based examples presented so far. For example, technical debt [3] [8] and software architecture quality often go hand-in-hand. An experiment by Izurieta et al. [23] shows forms of technical debt management occurring across a variety of software development phases. Given a strong connection between the management of technical debt, quality of a system, and the need to involve both in a quality control loop, the quality modeling concepts discussed in this paper could provide the modeling concepts needed for tangential applications.

An analysis of security vulnerabilities detectable by static analysis tools that associate with technical debt costs [22] aligns closely with software quality assurance indicating important application of quality modeling techniques in the security domain. Architectural quality assessment using measurements and prediction data–

[8]Software Quality Assessment based on Life cycle Expectations
[9]`https://www.sonarqube.org/`

such as those presented in [33]–or introducing empirical measurement data–such as how the number of developers can impact software quality [32]–also align with quality assessment using modeling techniques.

2.6 Critiques of Quality Modeling

While the models discussed offer beneficial approaches and helpful contributions to the domain of quality modeling, all models carry biases and flaws worthy of critique and concern. One basic critique is the lack of operationalization of many of the models due to lack of tool support or their implementation existing only for niche academic purposes. At a high level, critiques fall into four categories: weak measurement, over-design, output meaning, and practitioner distrust.

Weak measurement One of the more pressing issues with quality model research, weak measurement refers to the low number of measurements supported by tools, their lack of relevance, and the errors associated with their measurements [20].

Consider the SIG maintainability model [17]. SIG measures the characteristic *Changeability* as a function of two metrics, complexity per unit and duplication, where complexity per unit is evaluated using a slightly modified cyclomatic complexity formula and duplication is evaluated as the percentage of all code that occurs more than once in equal code blocks of at least six lines. The more complexity or duplication there is, the lower the score is for changeability.

Two questions arise in this context: (1) are there more measures–such as tool-based findings–that should also be considered when evaluating changeability, and (2) how much influence should the values of the measures have on the evaluation score of changeability? These concerns represent a threat to construct validity and become especially apparent when attempting to use a quality model on a language with little

tool support. One quickly finds a situation where there are too few measures to assert any quality value judgement with confidence.

Over-design Modern modeling attempts quickly fall into an issue of being too complex. As modelers attempt to decompose high-level ISO/IEC quality concepts into their subparts, a model can get out of control with too many layers, thousands of nodes, and tens of thousands of functional connections. Additionally the design of these models takes an unacceptably long time and lacks automation. As this complexity builds, human understandability of the model's mechanisms and the meaning of its output becomes lost,[10] and the model likely becomes over-fit or rife with design errors.

A middle ground in design complexity exists but is difficult to achieve. As a model becomes too complex, understandability is lost, but if the model is too minimal, the granularity to map which measurements affect which quality concepts is lost.

Output meaning What do the numbers mean? When a quality assessment value–for example, the *maintainability* node–returns *0.71*, what does that actually represent? Some quality model approaches translate ratio scale values into ordinal scales such as using a Likert rating system or grade letters F - A. Even more nuanced, some approaches use concepts like a benchmark repository to tune these ordinal scales according to a repository of representatively good examples of quality, but even then how can a stakeholder feel the benchmark repository accurately reflects their perspective of good and bad quality values. One practitioner's assessment score of

$$maintainability = 0.71 = B+$$

[10]Models derived from machine learning heuristics often have this problem due to the black-box nature of their internals.

may reflect a thoroughly satisfying state of the system while that state may be completely unacceptable for another. These numerical or ordinal values are pointless without context and hint that a quality value itself is inherently subjective and relative only to the stakeholder.

An example of the lack of continuity and output meaning is shown in a study comparing the output values of two popular software quality models, SQALE and Quamoco, when assessing the same software products [19]. Evaluating the inter-rater agreement between the two assessment approaches reveals output scores in remarkable disagreement. The study concludes there is not yet enough consistency between quality modeling approaches.

Beside the raw numerical meaning, the meaning of quality concepts such as *maintainability* or *usability* can be brought under question. An evaluation of the use and usefulness of the ISO/IEC 9126 standard is given by [1]. A key conclusion from the study is that the standard is too ambiguous in high level meaning, and is incomplete with measurement directives to be appropriate for quality assessment when software may put lives or businesses at risk.

Practitioner distrust Any combination of the above critiques leads to a failure in the pragmatic element of quality modeling: trust and beneficial use. Two surveys conducted with active industry subjects [42] [43] reveal a complete rejection to use preset quality models in their quality assessment. At a minimum, the models need to be tuned to the stakeholder's specific domain and conceptualization of quality; however, even then there exists a distrust of the values produced by the quality model.

2.7 Conclusions

Quality assessment through use of quality models is still a young field requiring significantly more research and validation before finding wider adoption in industry.

A key struggle with quality modeling attempts has been effectively bridging the gap between direct software measures and their impact on abstract quality concepts. Operationalizing quality assessment is also historically difficult due to the academic nature of the projects. The tool support for gathering metrics and findings data leads nearly all experiments to only evaluating Java language projects, and the tools themselves are often outdated and difficult to connect.

To pave the way forward, a foundational platform for research in quality modeling is needed that addresses the problems of weak measurement, over-design, meaning of output, and lack of trust with practitioners. The platform should facilitate ease in experimental academic use and industrial operationalization through modular design, simple input and output, easily understood internal mechanisms, awareness of inherent quality subjectivity, compatibility with continuous integration, and the ability to interface unknown future tools and languages.

To address academic rigor through a meta model and bridging the measures-concepts gap, section 3.1 reviews a recent quality modeling approach, Quamoco, which offers major steps towards these designs. Regarding practitioner trust and restraining over-design, section 3.2 reviews the QATCH quality modeling approach: a three-layer fully connected model which also offers a unique approach to injecting stakeholder subjectivity into the model without requiring major time investment or expert knowledge of the quality system.

The Quamoco and QATCH contributions are hybridized along with many new

contributions into the platform and focus of this thesis, PIQUE,[11] in chapter 4. PIQUE is used to generate experimental models in a relatively effortless manner to use for test cases and metric data generation.

[11]Platform for Investigative software Quality Understanding and Evaluation

CHAPTER THREE

SUPPORTING WORK

The PIQUE framework presented in this thesis is inspired and supported by the work of two recent quality model publications, Quamoco [10] [41] [40] and QATCH [37]. This chapter reviews the design and mechanisms of each approach to build the necessary technical knowledge for chapter 4.

3.1 Quamoco

The Quamoco project is specifically designed to bridge the gap between software measurements and high level quality concepts. The authors directly state as their research objective,

> "Our aim was to develop and validate operationalized quality models for software together with a quality assessment method and tool support to provide the missing connections between generic descriptions of software quality characteristics and specific software analysis and measurement approaches." [40]

Quamoco also includes a meta model for quality models [41], a generic and extendable base model, a modularized model approach, an assessment method integrated with the meta model, and benchmarking data for the base model.

Terms and Definitions

Quamoco has five main structural terms which follow the hierarchy shown in Figure 3.1.

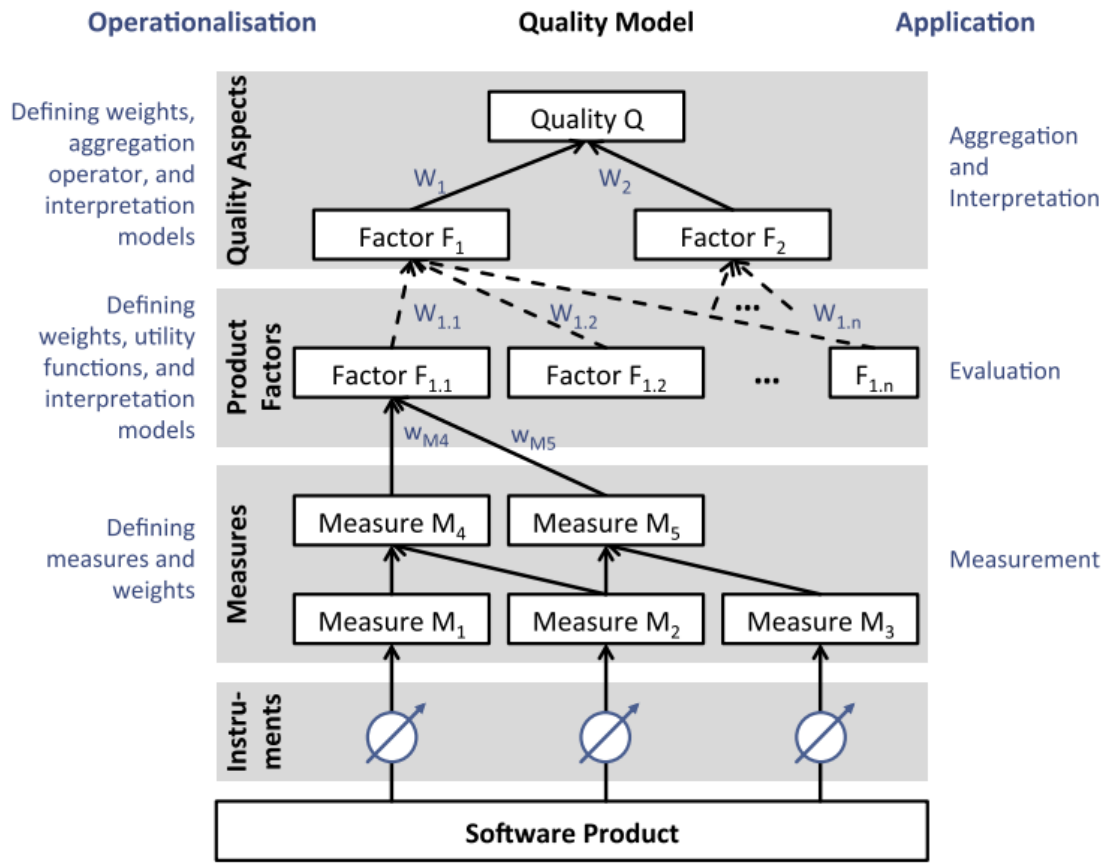

Figure 3.1: Quamoco quality assessment approach. Source: [40, p. 110]

Factor A factor is a high-level term which expresses a property of an entity such as the "cohesion of classes" or the "portability of the product". A factor is specialized into a quality aspect or a product factor.

Quality Aspect A quality aspect is a factor that expresses abstract quality goals such as the maintainability of the product. The quality decompositions given by ISO/IEC 25010 in Figure 2.1 are quality aspects.

Product Factor A product factor is a factor that represents the attributes of parts of the product such as the duplication of a source code part. Product factors at the

lowest level of the factor hierarchy must be concrete enough to be measured.

Measure A measure is a concrete definition of how a bottom-level product factor can be quantified through product-level measurements. A measure must decompose into something directly obtained by an instrument.

Instrument Instruments are the tools that audit and gather values from the software product. They are the concrete realization of a bottom-level measure.

Main Concepts

Bridging the Gap To bridge the gap between product measurements and quality aspects, Quamoco structures a relationship from product factors to quality aspects called an impact. An impact is a functional relationship that formalizes how the value of a product factor can translate into the value of a quality aspect. Because product factors must eventually be concrete enough to represent a direct measurement, and quality aspects express abstract concepts, this is an intuitive method to aggregate measures to abstract concepts. A figure of the impact bridge between product factors and quality aspects is shown in Figure 3.2.

Meta Model A quality meta model is a formal description of the elements and interactions a quality model can have. The existence of a meta model is useful for future quality modeling research and industry use of their actualization. It provides a common language for communication and understanding as different modeling approaches are attempted. Adherence to a meta model also allows confidence in the model's completeness and understandability if the meta model has been shown to be complete and understandable.

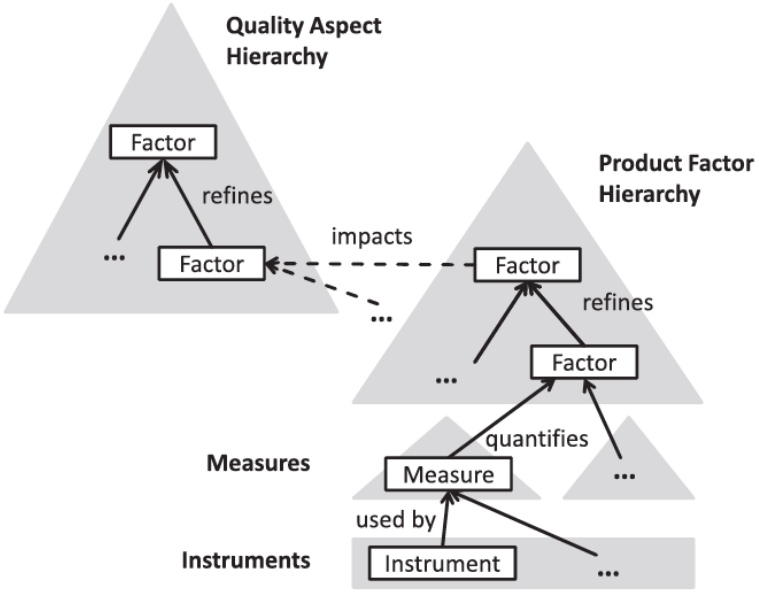

Figure 3.2: Quamoco structural concepts. Source: [40, p. 105]

The Quamoco meta model presented in [26] [41] helps with the implementation of quality models separate from the quality specification and quality evaluation elements. The UML for the meta model is shown in Figure 3.3 where elements above the horizontal line represent specification concepts and elements below the line are evaluation concepts. Empirical validation of the quamoco meta model is given in [26], showing the meta model as sufficiently general and satisfying in its understandability.

Modularity The final core concept of the Quamoco approach is its modularity. By designing the model with modularity and inheritance in mind, common factors for high-level concepts such as *object-oriented* or *GUI* can be moved to superclasses while leaving language-specific implementation details to the child classes. This eliminated duplication provides a common base of knowledge when considering factors for a given model and simplifies validation and experimentation on niche models.

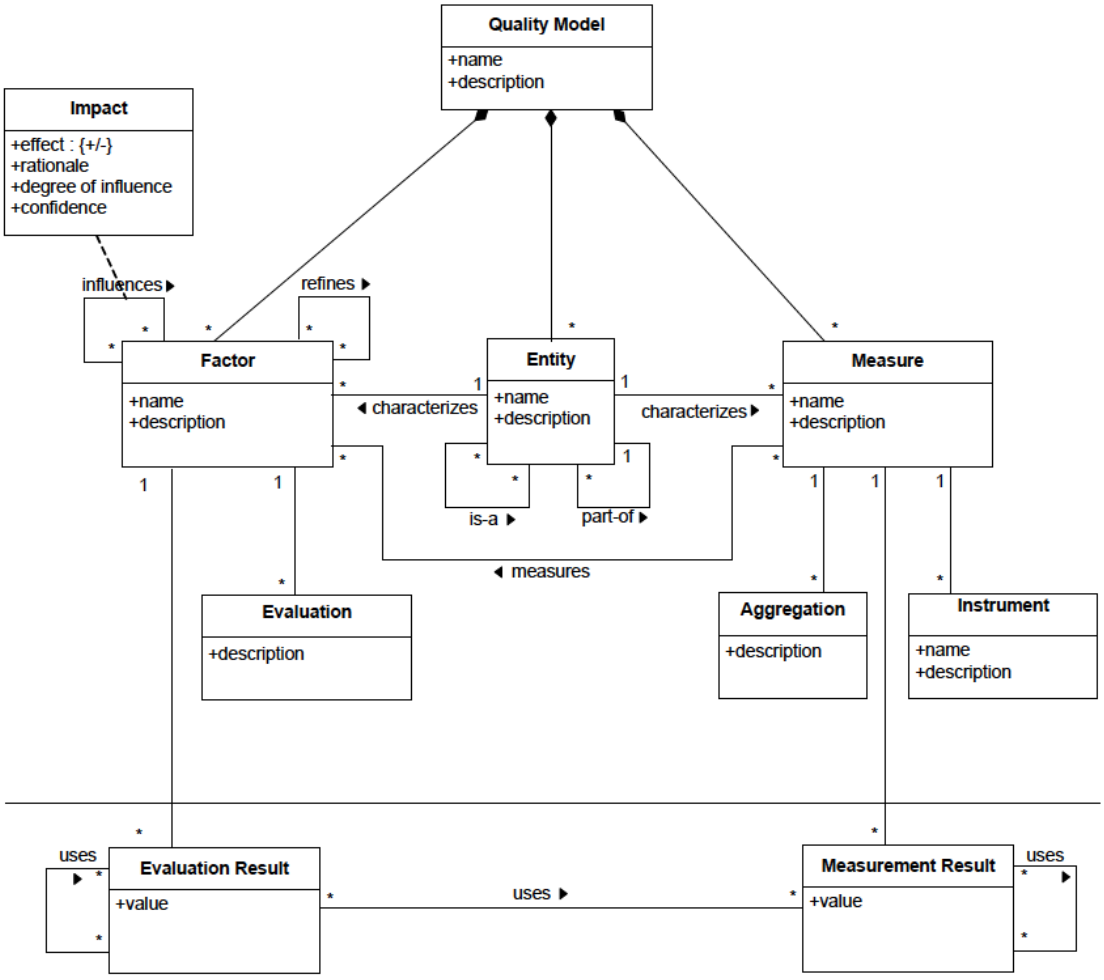

Figure 3.3: Quamoco meta model. Source: [41]

Mechanisms

Creation of a Quamoco model for use in product assessment, as shown in Figure 3.1, involves three inner mechanisms beyond simply defining a factor hierarchy and connecting measurements with tools: normalization, use of a benchmark repository for utility functions, and weighting of factor node incoming edges.[1]

[1]PIQUE implements these mechanisms as well with minor modifications, and the platform allows for the mechanism's functions to be overridden and modified (chapter 5).

Noramlization Quamoco normalizes by first defining a variety of bottom-layer measures such as *Lines of Code* and *Number of Classes* and then introducing a parent layer of normalized measures whose evaluation is defined by using the measure of interest and dividing it by the appropriate normalization measure.

Utility Functions Quamoco applies benchmarked utility functions to each measure directly used for evaluation of a bottom-level product factor. A simple increasing or decreasing linear function–similar to the utility function explained in section 2.3–is used on the measure against the benchmark data resulting in a value between $[0.0, 0.1]$.

Factor Edge Weighting Numerical relative importance weighting is applied to all product factor and quality aspect edges in the quality model hierarchy. Quamoco assigns weights using human-gathered importance orderings run against the rank-order centroid method [5]. These weights then evaluate up the factor hierarchy using weighted summation.

Quality Assessment Process

Quamoco quality assessment follows a method similar to most ISO/IEC hierarchal attempts using four main steps: measurement, evaluation, aggregation, and interpretation. These steps follow the right-side application steps shown in Figure 3.1.

Measurement The tools are first run against the product and used as input to the lowest level measurement quality model nodes. These values are used to start the evaluation chain up the measurement node hierarchy where normalization and other model-defined functions occur.

Evaluation The measure node values at the top of the measure hierarchy are transformed by the benchmark utility functions and used as input by the product factors at the bottom of the factor hierarchy. These product factors evaluate using weighted sums of their associated measure nodes.

Aggregation Aggregated evaluation of all factors up the factor hierarchy occurs using the edge weights provided by the model.

Interpretation Finally, the quality aspect ratio scale values need to be translated into values that are meaningful for human interpretation. The Quamoco approach [40] suggests mapping the values to an F - A grading scale. Any additional visualization techniques should occur at this stage.

Experimental Results

The Quamoco team ran validation tests on an operationalized Quamoco quality model by running experiments on the Quamoco base model against a variety of Java projects [40]. The research questions focused on whether the assessment results were valid, whether the model could be used to detect quality changes over time on a singular project, and if practitioners could find value in the quality assessment approach. The results were encouraging, and drove the decision for PIQUE to structure its model designs in a similar manner.

To determine the validity of assessment results, the Quamoco team compared how their model ranked five Java projects against expert judgement of the same five projects. The results are shown in table 3.1. Despite one difference in ordering, the Spearman's rho correlation of the rank ordering is $\rho = 0.9$ and a significant positive correlation is found.

Table 3.1: Quamoco base model results versus expert rankings [42]

Product	LOC	Quamoco Grade	Quamoco Rank	Expert Rank
Checkstyle	57,213	1.87	1	1
RSSOwl	82,258	3.14	2	3
Log4j	30,676	3.36	3	2
TV-browser	125,874	4.02	4	4
JabRef	96,749	5.47	5	5

The study evaluating trust in the model focused on perceived suitability and technological acceptance by practitioners in software development. While the questionnaires and interviews found the quality modeling approach more helpful to their quality assessment needs than simply using ISO/IEC 25010 definitions, the results were inconclusive regarding the validity of its output and whether the system would be useful to adopt. This result is a driving factor in the work of PIQUE, which prioritizes understandability and uses a different approach to interpretation of results.

Quamoco Conclusions

The work in the Quamoco project is a major stride forward in quality modeling research and its use in industry. It shows that ISO/IEC standards are a good starting point to interpret quality, but a model must extend the standard's definitions. In order to extend, a bridge is essential to connect measurements with abstract concepts. Quamoco's use of product factors impacting quality aspects is a good approach to creating this bridge. The project also shows that modular design is important to consider, and a quality meta model is useful for communication of mutually understood terms.

The project does have a large body of future work and concerns to address. The

experiences over the three year development of the base model show that validation of a model is an intensive and lengthy process due to the empirical and subjective nature of quality. Futhermore, model designs will have frequent problems and inaccuracies introduced over their development.

As a Quamoco model grows in size, the Quamoco model editor application provided with the project becomes too slow or unresponsive. Quamoco models also have a tendency to be too large and complex as shown by lack of understanding from practitioners not versed in quality modeling concepts. Designing a large model is also a long and arduous process. Given that it took the Quamoco team 23 people and 3 years, the authors conclude,

> "We are not sure to what extent such effort can be expended in practical settings." [40]

The industry interactions also reveals the need for a quality modeling expert to work with the stakeholder to correctly gather requirements and tune the model to their needs. Many companies would be unwilling to make such an investment. Instead, there should be a way to allow practitioners to communicate their quality expectations and influence the model design without needing domain-specific knowledge of the model.

Framework Concerns While the Quamoco base model, GUI quality model editor, and wizard-based quality adaption tools provided by the framework are useful for guided operationalization, interacting with the framework from a researcher's point of view is difficult and restrictive. This drives the motivation to provide a Quamoco inspired platform that facilitates open source contribution and experimentation in the domain of quality research. The system discussed in chapter 4 looks to provide such a platform.

3.2 QATCH

QATCH, the Quality Assessment Tool CHain, is a quality modeling project from 2017 by a team of three researchers from the Aristotle University of Thessaloniki. While the project does not follow a meta model description and lacks the validation efforts seen in the Quamoco project, QATCH specifically focuses on an automated way to derive quality models aware and responsive to the subjectivity of the stakeholder. Additionally, the model approach is designed with simplicity and transparency in mind. These two contributions warrant the model's evaluation and understanding.

Terms and Definitions

A generic QATCH model instance is given in Figure 3.4 presenting three main structural terms:

Characteristic A characteristic is defined as the abstract components of a system. This matches intuitively with the quality aspects of the Quamoco meta model. For example, the ISO/IEC 25010 components such as *Security*, *Maintainability*, or *Installability* are characteristics.

Property QATCH defines a property the same way the SIG maintainability model does: an inherent representational attribute of an object that can be directly measured [17]. These measurements can be either metrics or tool-based findings. A property generally refers to a property of source code such as volume, duplication, or structuredness.

Measure A QATCH measure follows a similar definition to a Quamoco measure: a concrete, quantified representation of tool's audit results.

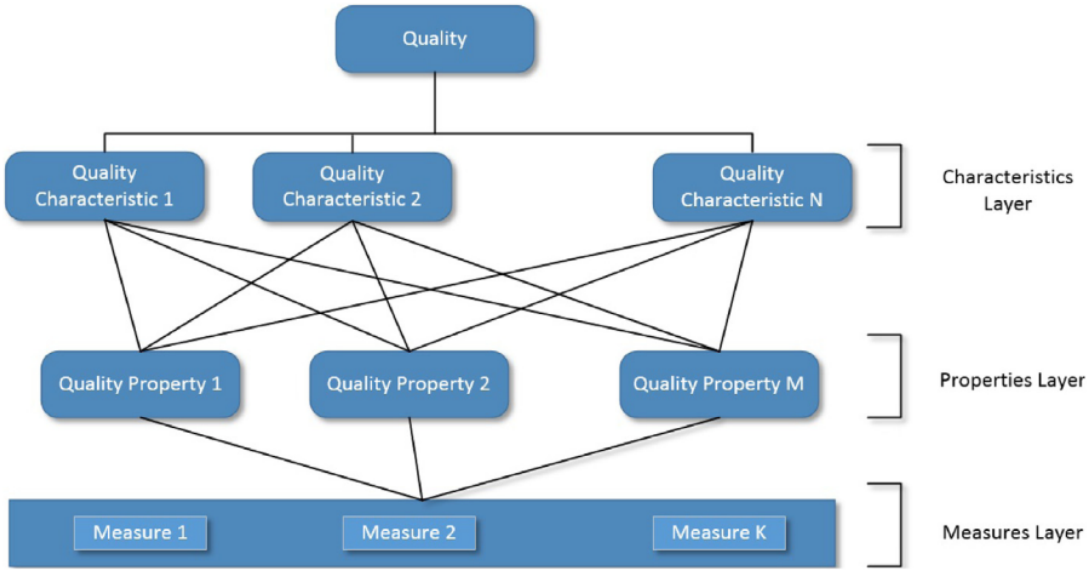

Figure 3.4: Qatch generic model instance. Source: [37, p. 363]

Main Concepts

Transparent, Understandable Model In order for a model to be trusted and used in industry, the model itself must be easily understood by non-experts. Understandability brings confidence which brings trust which brings use. The QATCH team understands this and built their model requirements with the following rules:

1. The model produced must not be derived from black box methods.[2]
2. The model must have only three layers.
3. Each property node is quantified by only one measure.

These design decisions are inspired from the same decisions used in [17]. While any design decision has its strengths and weaknesses, the QATCH authors [37] argue that

[2]For example, machine learning techniques.

the benefit of comprehensibility and ease of extension their model brings outweighs the loss of granularity brought by large, complex models.

Automated Subjectivity A concern from other quality modeling projects is the time that must be invested with a stakeholder to tune the model to the client's subjective interpretation of quality. The QATCH project addresses this with a two-fold approach: (1) Let the stakeholder express value judgements of quality concepts using intuitive terms while not requiring technical knowledge of quality modeling. (2) Integrate the value judgements into the model derivation process in an automated fashion.

The analytical hierarchy process (AHP) [34] is a method of using pairwise comparisons to derive a numeric ordering of importance. In the case of the QATCH quality model, the objects of comparison are quality concepts such as the characteristics *Security*, *Maintainability*, or *Installability* of system properties. The authors argue AHP can be used to elicit the model's weights because the quality model is a hierarchy.

Because pairwise comparisons can be expressed with linguistic values–*a is not important compared to b*–and it is more intuitive for a non-expert to compare two values instead of rank-ordering every quality concept, QATCH's use of AHP is a unique and valuable approach to eliciting weights to represent subjective opinion. The authors also present a fuzzy form of AHP that allows the practitioner to input their feeling of uncertainty for any given pairwise comparison.

Mechanisms

Similar to the Quamoco mechanisms discussed previously, the internals of QATCH need to address normalization, utility functions, and hierarchy edge weighting.

Normalization QATCH distinguishes its normalization method depending on whether the measure is of type metric or of type finding.[3]

As shown in equation 3.1, for any given metric, M, QATCH sums the value of the metric at each class of the system, m_i, times the lines of code of that class, LOC_i, and divides the sum by the total lines of code of the system, $TLOC$.

$$M = \frac{\sum_{i=1} m_i LOC_i}{TLOC} \tag{3.1}$$

QATCH normalizes a finding according to the number of occurrences of that finding weighted by its severities divided by the total lines of code.

Utility Functions Similar to Quamoco, QATCH uses benchmark repository data to define the utility functions. However, unlike Quamoco, QATCH applies thresholds with linear interpolation to the utility curve to achieve a more representative curve.[4] Furthermore, QATCH takes advantage of its model's requirement to have only one measure per property node and evaluates the utility functions at the property level instead of the measure level.

Hierarchy Edge Weighting QATCH uses the AHP or fuzzy-AHP process to weight the edges of its characteristic and property hierarchy. During model creation, the QATCH framework looks for a *.xls* file of manually-entered AHP values and returns an accordingly weighted model.

[3]Metrics and findings are discussed in chapter 2.2.

[4]PIQUE uses this same approach as its default utility function, and the concept is discussed in further detail in section 4.2.

Quality Assessment Process

Running a QATCH product assessment works similarly to a Quamoco assessment and most other ISO/IEC quality model implementations using a four-step process of measurement, evaluation, aggregation, and interpretation.

Measurement QATCH assessment calls the static analysis tools to gather measurement data. The data (metrics or findings) are sent to the appropriate measure nodes where normalization occurs.

Evaluation Using their connected measure node value as input, property nodes evaluate to the utility function output.

Aggregation These property node values then aggregate up one layer to set the values of the characteristic nodes using the weighted sums of their edges. The root node aggregates the same way using the weighted sums of its characteristic children.

Interpretation Finally, the root node ratio scale value is given a one- to five-star rating where one star is a value $[0.0, 0.2)$ and five stars is a value $(0.8, 1.0]$.

Experimental Results

To assess quality model result validity, the QATCH team ran an experiment identical to the Quamoco validity experiment discussed previously. The same five Java projects were evaluated using an operationalized QATCH model, the quality score rankings of the five projects were compared against Quamoco's ordering and the experts' judgement rankings. The results are shown in table 3.2. The statistical study of [37] shows a perfect correlation between QATCH's ranking and the experts'

Table 3.2: QATCH model results versus Quamoco and expert rankings [37]

Product	QATCH score	QATCH Rank	Quamoco Rank	Expert's Rank
Checkstyle	0.57201	1	1	1
Log4j	0.47829	2	3	2
RSSOwl	0.47490	3	2	3
TV-browser	0.47466	4	4	4
JabRef	0.45239	5	5	5

ranking, and a significantly positive Spearman's correlation of $r = 0.9$ between QATCH and the Quamoco results.

QATCH Conclusions

The QATCH project offers many ideas and contributions worth integrating in future quality modeling research. The use of AHP [34] is an especially strong idea that allows non-technical practitioners to use non-technical language to inject subjectivity into a quality model without much time or effort. Requiring a model to have only three layers is a good alternative given that one of the biggest critiques of other models is practitioner understandability and trust; however, only three layers may be too restrictive on model effectiveness. More research is needed in this area. The use of thresholds and linear interpolation to the benchmarked utility functions is also a valuable way to add more precision without significant loss of understandability.

System Construct and Mechanisms Concerns Concerns do arise regarding the internal mechanisms and construction of QATCH. Unlike Quamoco, The QATCH model does not intentionally adhere to a meta model, and it was not designed with modularity in mind. This necessitates more validation of the project and also causes

experimentation to be more difficult due to the lack of modular structure.

While it is possible to extend the model with more properties or tools, many internal parts of the measurement and evaluation process are directly tied to two Java tools, CKJM[5] and PMD[6], and parts of the model derivation process are hard-coded to array index references. While the project works well for out-of-the-box assessment and basic experimentation, the design does not facilitate major extensions to the system such as language support beyond Java.

3.3 Conclusions

Quamoco [40] and QATCH [37] are two modern quality assessment attempts. They correlate with each other and experts' opinions on ordering Java projects by their quality, and they attempt to address the natural subjectivity of evaluating quality. The analysis of these two approaches and the review of historical quality modelings (chapter 2) reveals gaps that need addressing.

Quamoco and QATCH show that extending the ISO/IEC standards is likely the right approach, and one must figure out a way to bridge the gap between low-level measurements and the abstract concepts of ISO/IEC. While attempting to bridge the gap, these projects show that building quality models is a burdensome and lengthy task. The models are difficult to operationalize, difficult to validate, difficult to extend by other researchers, and quickly become outdated and no longer supported. Even when a model is operationalized, practitioners tend to not understand their mechanisms nor trust their output values.

[5] `http://http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/`
[6] `https://https://pmd.github.io/`

What do the Numbers Mean?

Experimentation with Quamoco and QATCH reveals differences in scoring. Both projects were run on five Java projects: Checkstyle, Log4j, RSSOwl, TV-Browser, and JabRef. The QATCH quality scores and Quamoco scores on the five projects is shown in table 3.3.

Table 3.3: QATCH vs Quamoco scoring of the same products [42] [37]

Product	QATCH Grade [0, 1]	Quamoco Grade [6, 1]
Checkstyle	0.57201	1.87
Log4j	0.47829	3.36
RSSOwl	0.47490	3.14
TV-browser	0.47466	4.02
JabRef	0.45239	5.47

Even when both QA techniques are used on the same system, we see differences in the range and slopes. Quamoco's range in scores cause Checkstyle to appear to have remarkably better quality than JabRed, but QATCH's score range causes all five projects to appear approximately equal and mid-range in quality score. Thus, even if two different quality models value-rank projects with high correlation, there is a problematic lack of meaning in the numerical values output for any single project. This is a problem that must be addressed as practitioners often want to see the underlying numbers a model is producing.

A Platform for Quality Modeling Research

Additional research in quality models is needed before its wide-spread adoption in industry. The validation and adoption of advanced techniques in quality assessment–such as the use of quality models–has never been more needed because

a failure in quality in software could lead to loss of life, economic ruin, and threats to national security. This calls to action the need for a platform to assist in quality modeling research.

The analysis of Quamoco and QATCH serves to guide design concepts for what a platform intended for quality model research should support and restrict. Is there a way to reduce the time spent conceptualizing and operationalizing quality models so more time can be put on validation and industry feedback? Can the platform facilitate the experimental needs of a researcher by allowing simple extension of mechanisms such as different normalization techniques, new static analysis tools, other utility functions, and new node evaluation approaches? Is there a way to bring meaning to the values output by a model apart from representing it as an ordinal scale? Can the platform support experimental visualization output techniques?

CHAPTER FOUR

PIQUE SYSTEM DESIGN AND TECHNICAL DETAILS

PIQUE, a Platform for Investigative software Quality Understanding and Evaluation, is a platform that can be used to create an environment for further software quality assessment research. The platform and an operationalization using the platform are built to generate the objects needed to run test cases to address the research questions and goals given in chapter 1.2.

Design goals of PIQUE are listed in table 4.1. These goals come from reviewing the strengths and weaknesses of the previous modeling approaches presented in chapters 2 and 3. Through test cases, exercises, and discussions provided in chapters 6 and 7, each of the following design goals has been partially or fully accomplished.

Table 4.1: PIQUE design goals

Design Goal	*Motivation*
DG01 - Benchmarking, utility functions, and adaptive edge weighting supported	Modern modeling research [40] [37] shows that benchmarking, utility functions, and adaptive edge weighting are necessary mechanisms to utilize in the derivation of a quality model.
DG02 - Default model mechanisms	By providing default classes for model derivation, researchers can quickly derive and operationalize a model without needing to override and understand every mechanism.
DG03 - Extension or modification of model mechanisms	To facilitate experimentation, each component of model derivation should be overridable.

DG04 - Models are easy to derive	The framework should automate as many aspects of model derivation as possible. Running derivation should be an easy to understand process.
DG05 - Derived models are easy to operationalize	The framework should provide a way to operationalize its models with as few steps and dependencies as possible.
DG06 - Adding, removing, or modifying tool support is simple	Analysis tools and programing languages can evolve rapidly and are the foundational component of quality model assessment. The framework should facilitate straightforward interfacing with unknown external tools.
DG07 - Input and output is easy to interact with	A text-based format should be expected for both input and output to allow the framework to work with any operating system or visualization tool.
DG08 - Facilitate automation and continuous integration	The framework should be designed to facilitate automation and CI in order to achieve the pragmatic elements of its intended use. The generated models are more likely to be used by practitioners if there is little overhead to integrate assessment into their development cycle.
DG09 - Facilitate trustable models	Ultimately, the models produced by the framework must be understandable and trusted by both researches and practitioners. This can be accomplished by limiting complexity and allowing complete views of all layers of the model.

DG02 and *DG03* express an important duality needed in the platform design: there should be a near out-of-the-box state of the system to help new researchers need

as little time as possible to accomplish a "hello world" run of model derivation and product assessment, but the platform should also allow for complex extension and modification of its mechanisms to drive research and exploration.

4.1 System Design

The PIQUE system is designed as a collection of library functions and API features. It is an open-source project written in Java hosted at the Montana State University Software Engineering Labs (MSUSEL) public GitHub.[1] The project uses Maven[2] for dependency management and system building, testing, and packaging. A verbose README exists in the repository to provide new users or contributors an easy process toward use of the code base.

PIQUE takes the responsibility of implementing all functionality not directly tied to language specific tasks while leaving the researcher the requirement of connecting their language specific tools to the model measures. For clarity, the language agnostic system of library and API calls will be referred to directly as "PIQUE" while a language-specific operationalization of PIQUE will be referred to as "PIQUE-EXT".[3]

High-level View

PIQUE supports two processes needed for quality control: quality model derivation and product quality assessment. To communicate a general sense of how the PIQUE framework is intended to be used, Figure 4.1 shows a high-level view of the components in the context of a researcher deriving and using a Python-intended quality model utilizing the platform. The researcher will create a new project, in this

[1] `https://github.com/msusel-pique/msusel-pique`

[2] `https://maven.apache.org/`

[3] In the GitHub codebase for this project, the project called PIQUE is the project of library and API calls while a project named PIQUE-CSHARP uses the functionality provided by PIQUE to derive and operationalize a model that can assess the quality of C# .NET projects.

case named PIQUE-PYTHON, and implement the `ITool` interface. Implementing the interface requires the researcher to define how to run their Python static analysis tools and define how to interpret the tool's results as PIQUE class objects. The `Runner` interface is then utilized by passing in the researcher's Python quality model,[4] a directory containing the benchmark repository Python projects, and a file with the hand-entered AHP[5] values.

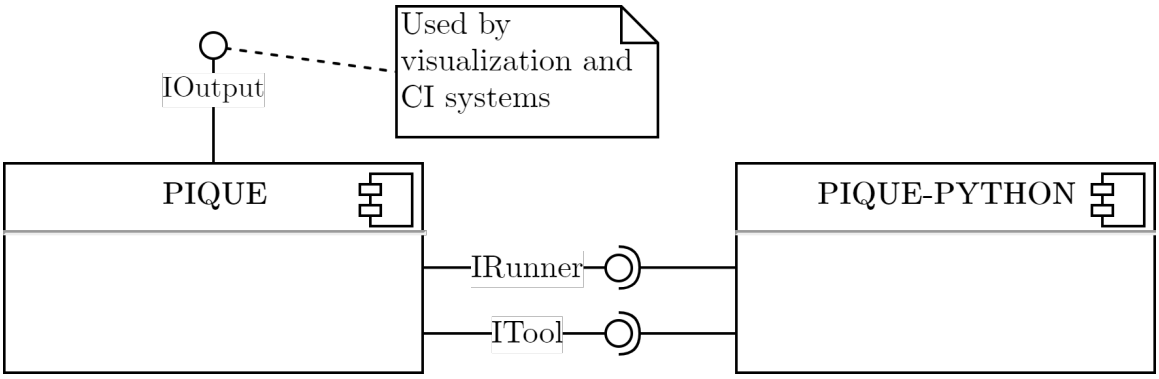

Figure 4.1: PIQUE operation high-level component view

This process outputs a functional quality model capable of assessing software systems written in Python. The researcher can then use another function in the `Runner` library to run quality assessment on a Python system by passing in the derived quality model and the directory of the system under assessment. For a detailed example of the process taken to derive and operationalize a PIQUE quality model, section 5.3 presents a use case for a quality model targeting a C# system.

It is important to recognize from this process how simple model derivation and assessment is when using PIQUE's default mechanisms. As long as the researcher defines (1) how to run the language-specific tools and (2) defines, as *.json* text configuration, how tool results connect to measurements, the components of PIQUE

[4]At this point, the input Python quality model represents the desired factor structure, but does not have utility functions, edge weights, or node values.

[5]Analytical Hierarchy Process. Introduced in chapter 3.2.

and its `Runner` library call handle everything else.

Model Terms and Definitions

Because the models output by PIQUE are designed with the concepts offered by the Quamoco meta model in mind, many of the same terms and definitions are used.

Factor A factor is the abstract term for nodes in the top layers of the quality model. Following form from Quamoco definitions, the following definitions of TQI, quality aspect and product factor are all of type factor.

TQI (Total Quality Index) TQI, a type of factor, refers to the the root node of the model–the quality topic under evaluation. In the context of ISO/IEC 25010 [14], TQI represents *Software Product Quality*, but it can represent whatever concept the researcher desires, such as *Security*.

Quality Aspect Quality aspects are high-level factors that express abstract quality goals that cannot be measured directly.

Product Factor Product factors are factor nodes that can decompose into directly measurable concepts. Similar to the Quamoco definition, a good way to come up with a product factor is to think about the attributes of the parts of a product. Following the concepts of QATCH, PIQUE by default fully connects each product factor to every quality aspect; however, this can be overridden through configuration in the quality model description file.

Measure Measures are concrete definitions of product factor values. A measure holds knowledge of its relevant benchmarked utility functions and contains the evaluation information needed to calculate its value from incoming diagnostics.

Diagnostic A new term introduced with PIQUE, a diagnostic is a representation of the parts needed for a measure to evaluate. A diagnostic must evaluate directly from the results of its connected tool's output, so a diagnostic is tool-specific (and thereby often language-specific) and defines what tool is used to obtain its value.

Consider a measure *Path Coverage* which is evaluated by dividing the number of routes executed by the total routes possible. Two child diagnostics of the *Path Coverage* measure exist in this scenario: (1) total routes possible and (2) actual routes executed.

Again, consider a measure *Cryptography Vulnerabilities* and a static analysis tool named Security Code Scan. Security Code Scan has the ability to find two cryptography-related findings, *SCS0004 - Certificate Validation Disabled* and *SCS0005 - Weak Random Generator*. If a researcher defined the *Cryptography Vulnerabilities* measure to be evaluated as the sum of findings of *SCS0004* and *SCS005*, then *SCS0004* and *SCS0005* are diagnostics connected to the measure.

The use of diagnostics allow the quality model designer to configure a measure in terms of the behavior of its parts. This is especially useful regarding exploratory measure definitions coming from static analysis tool findings. Consider a security example with the *Cryptography Vulnerabilities* measure used earlier. Say a hypothetical cryptography finding, *SCS9999 - Trigger Server Room Explosion*, is supported by a static analysis tool. Regardless of all other cryptography findings, the researcher realizes an evaluation function at the measure level needs to be defined:

```
if numFindings(SCS9999) > 0 then
    finalValueOf(cryptography_vulnerabilities) = 0.0
else ...
```

This evaluation function implies that–no matter how good or bad other findings

are in support of the cryptography score–if even one finding of *SCS9999* occurs, that is exceptionally bad because that piece of code could cause one's server room to explode, which represents a problematic quality concern. Therefore, the value of the *Cryptography Vulnerabilities* measure should immediately be zero if even one *SCS9999* finding exists regardless of the impacts from other child diagnostics.

In response to historical quality modeling attempts that handle tool finding evaluations as simply a count of their findings, the added evaluation possibilities diagnostics offer provide a valuable way forward to assessing the quality of difficult concepts such as security.

Finding A finding is the data object representation of a "hit" from its associated diagnostic and tool. A finding is only instanced after its associated tool has run an audit on the system under evaluation.

Quality Model Description

PIQUE design follows the metaprogramming recommendation of "Configure, Don't Integrate" from Hunt's *The Pragmatic Programmer* [18, p. 144]. Because a primary design goal of the platform is to facilitate experimentation, it is a sensible choice to leave as much manipulation and specification as text files used as input instead of requiring modification and recompilation of the code base.

For model derivation, a quality model description, AHP matrices, and the location of the benchmark repository are expected as input. For project quality assessment, a derived quality model is expected. This section will describe the text-based model configuration and its derived output format. AHP matrices and the benchmark repository are reviewed in the following section about PIQUE default mechanisms.

Table 4.2: PIQUE quality model configuration field descriptions

Node	Field	Example Values	Opt.	Default
n/a	global_config	Benchmark_strategy	yes	
factor	name	Total quality, Portability	no	
	parents	[null, Security]	yes	Fully connected
	eval_strategy	Weighted_sum, Largest_child	yes	Weighted sum
measure	name	Path coverage	no	
	parents	[Documentation]	no	
	positive	True, False	no	
	eval_strategy	Average, Zero_if_exist	yes	SUM
	norm_strategy	None, LoC, ClassLoc	yes	LoC
	diagnostics	[SCS0005, Diagnostic_obj_2, ...]	no	
diagnostic	name	SCS0005	no	
	description	Weak random number generator	no	
	toolName	Security_code_scan	no	
	eval_strategy	Count, Finding_value	yes	Count

A quality model description is a *.json* text representation of the model to be

derived. Table 4.2[6] shows the quality model description components and an example *.json* quality model description file is provided in appendix A.

Because the model is not yet derived, a quality model description does not need to give values for factor weights or measure thresholds. However, the quality model description does need to provide all configurations for its measures and diagnostics as well as optional factor hierarchy connections.

A model is designed through this description by defining the factors, measures, and diagnostics as *.json* objects. The model generation engine assumes all factor layers are fully connected, but a factor node can be defined as strictly child to another factor node if desired through use of a *parents* array. At the measure level, each entry is defined with a description, a positive or negative boolean value, an array of the measure's parents, and a collection of all diagnostic objects used to evaluate the measure.

Quality Model

After running model derivation, PIQUE returns a derived quality model in *.json* format, now with edge weights representing subjective value and utility functions representing expected ranges of measure values. If the quality model description of Figure 4.2 was run through a model derivation process, it can return with the form shown in Figure 4.3. The factor weights and utility functions are highlighted in yellow to show the changes to the model after running derivation.

Components

PIQUE is designed to facilitate model derivation and product quality assessment using quality models in a language-generic way while supporting default or modifiable

[6] [Square brackets] represent optional multiple values.

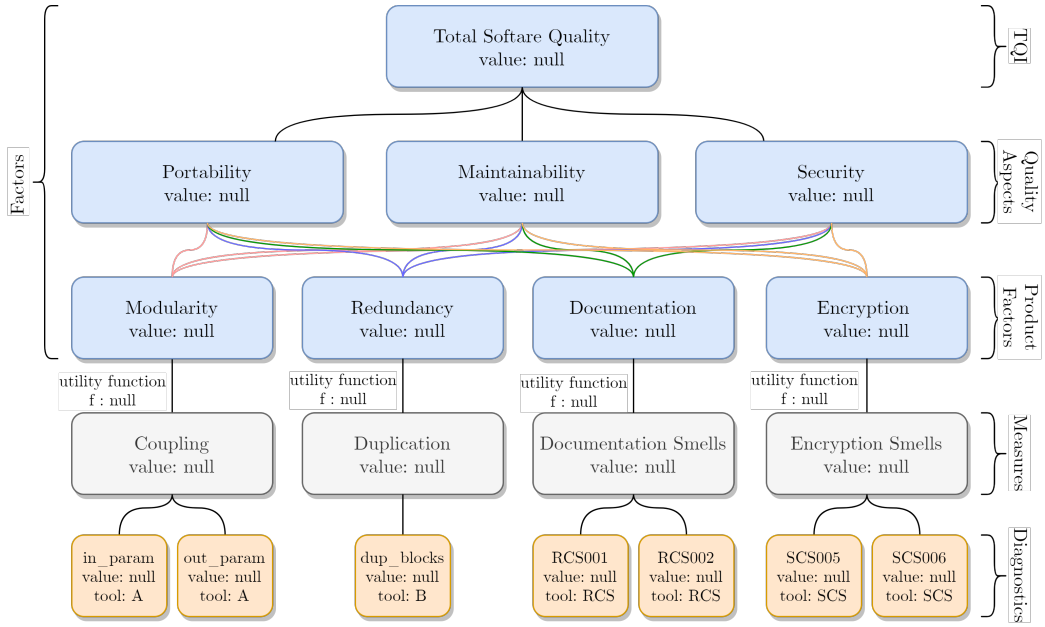

Figure 4.2: Quality model description: model view

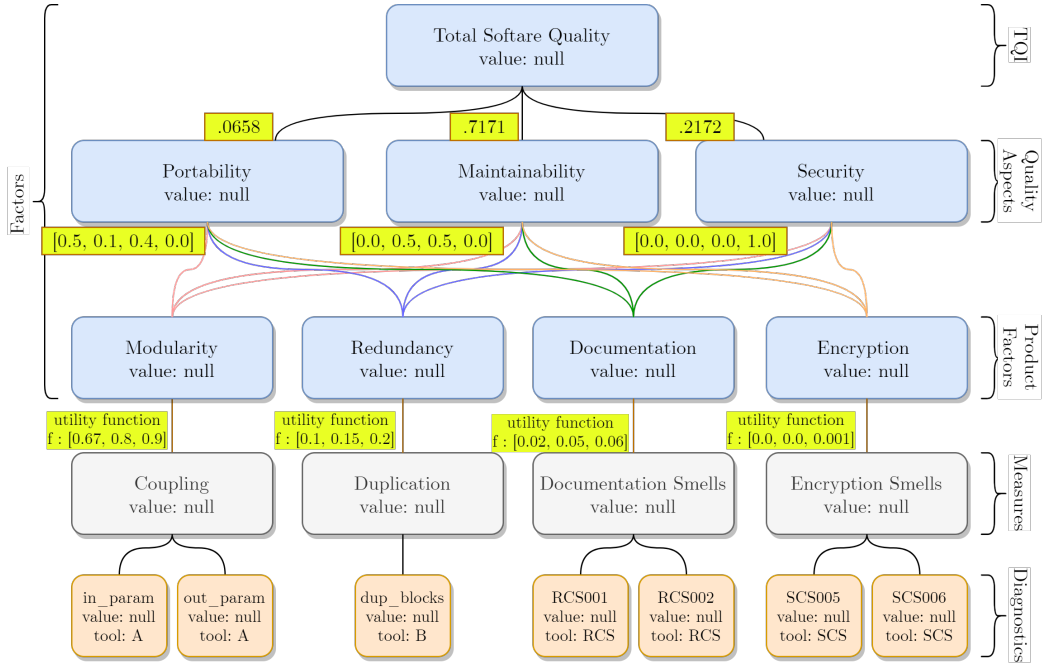

Figure 4.3: A derived quality model. Utility value thresholds are contrived for simplicity.

inner mechanisms. To achieve this, modular and orthogonal design is key. Throughout its development, the system evolved into the five components shown in Figure 4.4: Analysis, Calibration, Evaluation, Model, and Runner. This section provides a brief, technical review of each component.

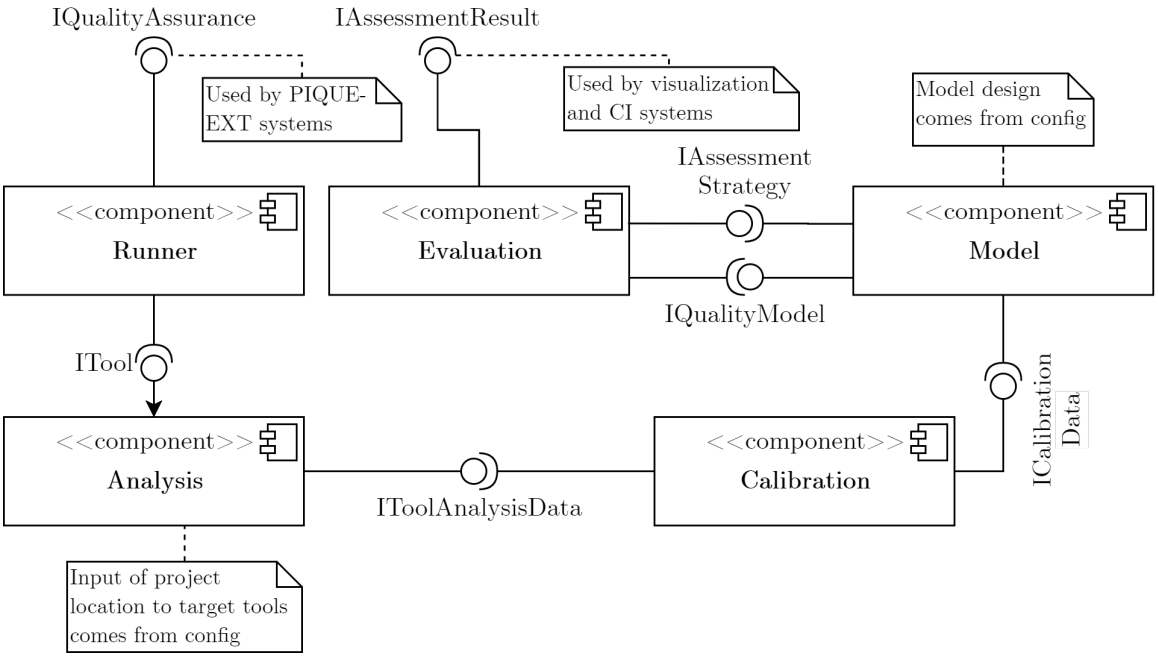

Figure 4.4: PIQUE component diagram

Model The Model component, shown in Figure 4.5 handles the construction and use of a quality model. By requiring model configuration and injecting non-default mechanisms through the configuration file, this component serves as the bridge between the configuration details and the system abstractions.

The Model component contains the `QualityModel` class which has a collection of `ModelNode` objects, a tree data structure. All model nodes have a value, a collection of zero or more children, a mapping of each incoming child's edge weight, an associated evaluator and normalizer object, and a method defining how the node should evaluate its value. By default, each node is assigned a default assessment strategy, but each

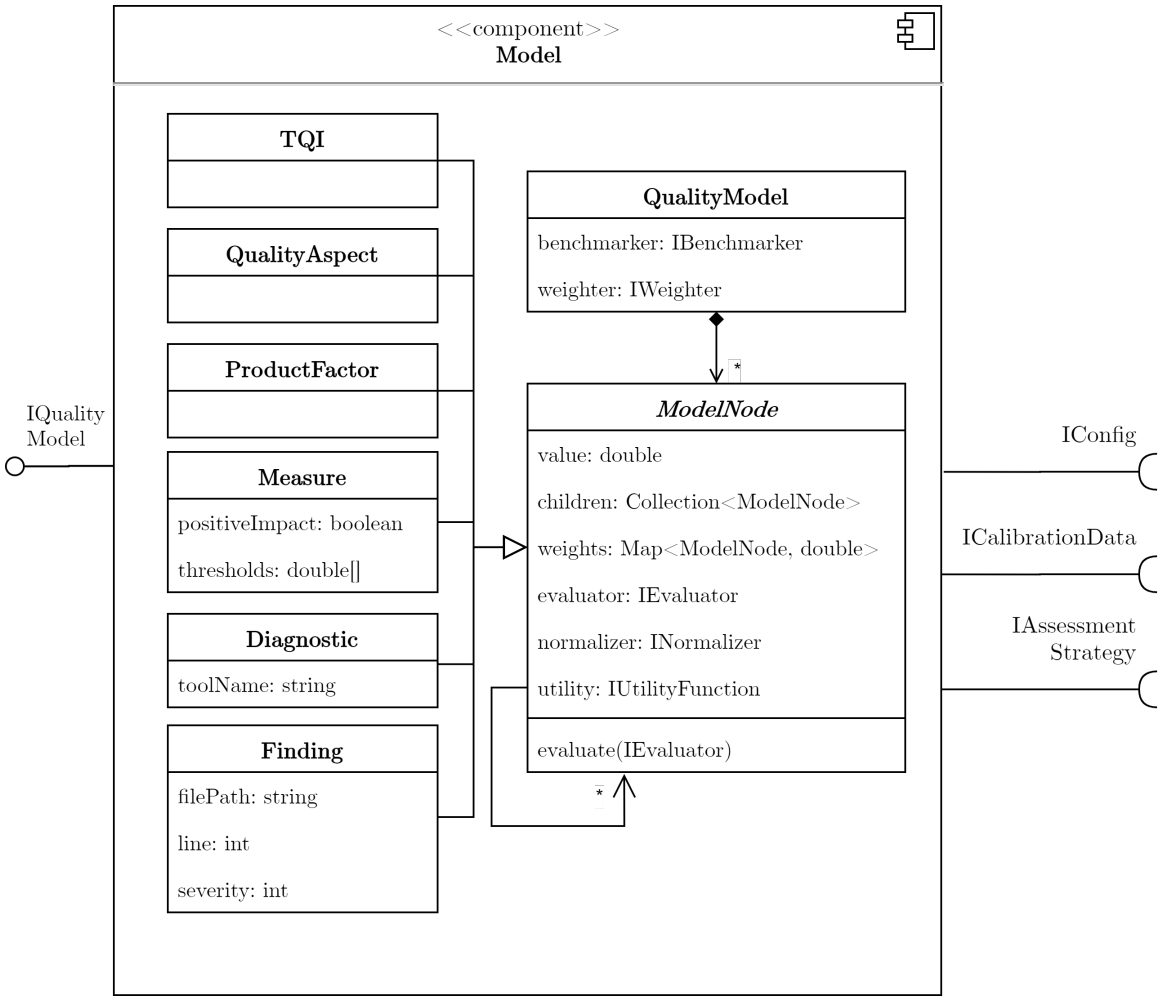

Figure 4.5: PIQUE Model Component

strategy can be overridden in an extending, language-specific PIQUE project and injection via the quality model description file. For example, a `QualityAspect` node will have a `DefaultEvaluator` evaluator associated with it by default which evaluates the node's value to be the weighted sum of its children's values, but this can be overridden by creating, for example, a `SecurityEvaluator` class which evaluates a quality aspect node's value according to a niche set of security approaches.

The MODEL component exposes an `IQualityModel` interface which can be used for tasks such as evaluation. `BenchmarkData`, `AnalysisData`, and `AssessmentStrategy`

interfaces are associations needed to construct the `QualityModel` tree object during model derivation.

Analysis The ANALYSIS component is responsible for static analysis tool processes such as activating a static analysis tool and interpreting its results. Because static analysis tool support or definitions can change at any given time, loosely coupled design of the analysis component is crucial.

The ANALYSIS component is simple, containing one interface named `ITool`. All language-specific tool classes must implement the `ITool` interface which contracts them to define how they run, how to parse their tool's result files, and how to transform their parsed data into a `Diagnostic` object. Figure 4.6 presents the high-level view of the PIQUE interface contained by the ANALYSIS component.

The `analyze(Path)` method activates the static analysis tool to run static analysis on a project location at `Path`. The return value is the location of the static analysis results produced by the tool.

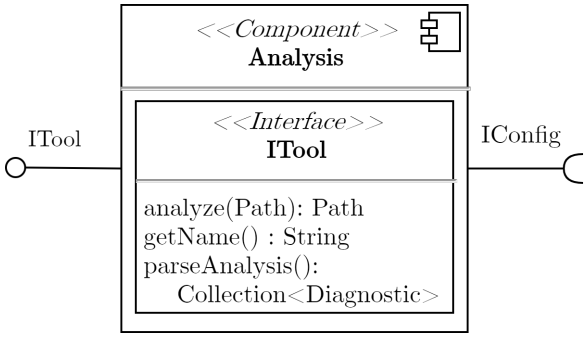

Figure 4.6: PIQUE Analysis Component

The `parseAnalysis()` method is the method responsible for transforming the tool result data into PIQUE domain objects, specifically `Diagnostic` objects. The method parses through each result line in the tool's output file and creates a new diagnostic object for each diagnostic supported by the tool, and each diagnostic is filled with a collection of its diagnostic-specific findings. This collection of diagnostics and composed findings are used by the CALIBRATION component to derive thresholds and by the MODEL

component to populate the layer of findings specific to the system under evaluation.

Calibration The CALIBRATION component is responsible for the procedures necessary to derive thresholds and factor edge weights. By separating model functions from calibration elements, modifying calibration mechanisms becomes a much safer task. Utility classes such as `IBenchmarker` and result abstractions such as `WeightResult` are found here.

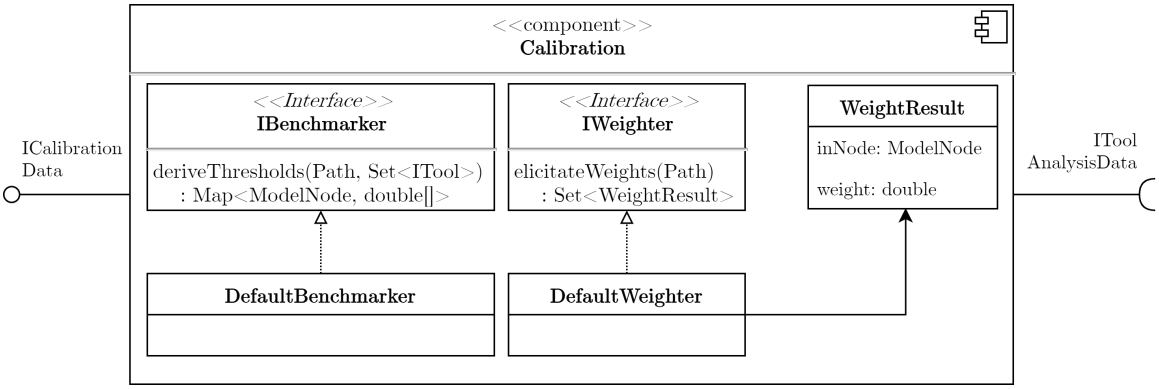

Figure 4.7: PIQUE Calibration Component

Calibration is primarily associated with the ANALYSIS component due to its dependency on the `ITool` object functions to receive static analysis data about the system under evaluation. This data–for example, information about diagnostic and measure evaluation across a repository of benchmark projects–is used for benchmarking and edge weighting procedures.

Evaluation EVALUATION distinguishes the quality model behavior from the quality model structure. This component handles the processes related to quality model assessment such as aggregation, normalization, and use of other evaluation strategies. These interfaces are intentionally kept separate from the quality model object in order achieve a separation of concerns and, thus, more effective extensibility.

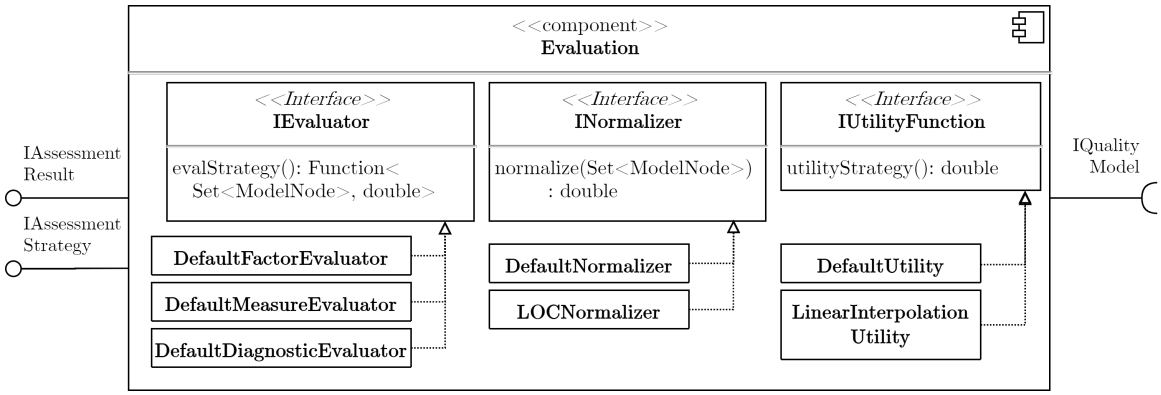

Figure 4.8: PIQUE Evaluation Component

An `IEvaluator` class represents a first order function that defines how a model node should evaluate given a collection of other model nodes. The default factor, measure, and diagnostic evaluators are instanced in PIQUE to provide a default method of evaluating quality model nodes to achieve a simple 'out-of-the-box' solution to quality model derivation or assessment. This same concept of default classes applies to the `INormalizer` and `IUtilityFunction` interfaces. The `DefaultFactorEvaluator` evaluates a factor node as the weighted sum of its children. The `DefaultMeasureEvaluator` evaluates a measure node to be the total sum of its diagnostic children, as long as the diagnostic is not the *LinesOfCode* diagnostic. The `DefaultDiagnosticEvaluator` evaluates a diagnostic node to be the total count of its child findings.

The `INormalizer` interface contracts a `normalize()` function that normalizes a model node object given a set of other model nodes. For example, the `DefaultNormalizer` is defined to apply no normalization by simply returning a model node's value. All factors nodes are, by default, given the `DefaultNormalizer` because their values likely do not need further normalization. On the other hand, all measure nodes are, by default, given the `LOCNormalizer` which is defined to divide the value of the containing measure node by the *lines of code* diagnostic, thereby normalizing

each measure by the lines of code in the system. This default should be extended if a model designer wishes to, for example, normalize some measures according to a class's lines of code.

The `IUtilityFunction` interface provides a `utilityStrategy()` function that applies a utility function to a model node. By default, all model nodes except measure nodes contain a `DefaultUtility` function which simply returns the node's input value. Measure nodes, by default, contain a `LinearInterpolationUtility` function that applies linear interpolation to the node's threshold value as described in section 4.2.

Runner The RUNNER component handles the interfacing, abstractions, and procedures necessary to allow language-specific extensions of PIQUE to simply use its `deriveModel()` and `evaluateProject()` functionality without worrying about implementation details as long as the ITool and Config contracts are fulfilled. This component is important to achieve design goals DG02 - "Default model mechanisms", DG04 - "Models are easy to derive", and DG05 - "Derived models are easy to operationalize" by removing coupling between PIQUE-EXT systems and the internal mechanisms and components of PIQUE.

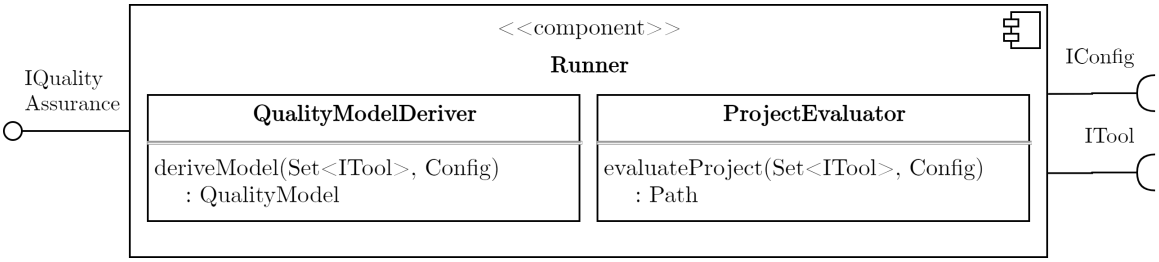

Figure 4.9: PIQUE Runner Component

Consider a language-specific C# extension of PIQUE named PIQUE-CSHARP. At a minimum, PIQUE-CSHARP will contain one or more classes implementing the ITool interface for each needed C# static analysis tool. If further modification

is desired, PIQUE-CSHARP can also contain classes extending the `IEvaluator`, `INormalizer`, and `IUtilityFunction` interfaces. Finally, a *.json* quality model description file is created which defines the quality model structure and, if desired, the associated behavior interfaces. These interfaces and a configuration file (containing information such as the location of the quality model description and the location of the benchmark repository) are used to obtain a derived C# quality model by running the `deriveModel()` method. In similar fashion, the `run()` method is used to run product quality assessment by passing it the set of necessary C# `ITool` objects, a configuration containing the quality model *.json* file (generated from the derivation process), and the path to the product under assessment.

Connecting Tools

A quality model is only as strong as the tools that support it. These tools–static analysis tools in the scope of this thesis–come from a variety of sources: third-party, proprietary, language-integrated, etc. Simply getting a static tool to run on a target project and store the results on disk is often the first hurdle a quality researcher will find themselves struggling with.

Because PIQUE is designed to be language-agnostic, the framework leaves the language-specific PIQUE-EXT project responsible for defining how to start up its tools, run its analysis procedure, and map the results to PIQUE domain objects. This is accomplished by implementing the ITool interface which requires by contract defining the `analyze()` and `parseAnalysis()` methods.

4.2 Default Mechanisms

As identified through analysis of previous quality modeling attempts, the core mechanisms needed for quality model derivation and use are normalization functions,

utility functions via a benchmark repository, factor edge weighting via subjective opinion, and aggregate evaluation functions of the model nodes. As shown in section 4.1, PIQUE provides default functionality for these mechanisms that take inspiration from the Quamoco [40] and QATCH [37] projects.

Normalization Functions Normalization occurs at the measure level before utility functions are applied to the measure values. Some measures should be normalized, but not all. In a general intuitive sense, many metrics do not need to be normalized as they often represent system-scoped values, but most findings should be normalized due to the size of the system directly influencing the number of findings that occur.

PIQUE requires a tool connection that returns the lines of code (LoC) in the system under evaluation. Then, by default, every measure is set to be normalized by LoC. If a measure should not be normalized–such as coupling–or a different normalization technique should be used, a measure can be configured using the quality model description configuration.

Utility Functions and the Benchmark Repository As discussed in section 2.3, a collection of benchmark projects provide the information needed to generate utility functions at the measure layer. When PIQUE is provided a directory of well-formed benchmark projects,[7] calibration mechanisms provided by the framework complete the rest of the work needed to apply utility function data to the model's measure nodes.

PIQUE's default benchmarking follows the same approach used in the QATCH project [37], which uses the threshold formulas provided by [29]. For every measure

[7]The file structure of each project needs to be structured in such a way that the static analysis tools used for auditing will run correctly when iterating through a collection of systems. Finding better solutions than requiring well-formed structure is saved as future work.

defined by the quality model description, the benchmark repository is used to derive three thresholds, t_1, t_2, t_3, representing the minimum, median, and maximum values respectively of the measure after removing outliers. Specifically, this is defined by equation 4.1 where x_i is the normalized value of measure x for benchmark project i where $i = 1..n$, Q_p is percentile p, and IRQ is the inter-quartile-range.

$$
\begin{aligned}
t_1 &= min(\{x : x \geq Q_{25\%} - 1.5 * IRQ(x_1, \ldots, x_n)\}) \\
t_2 &= median(x_1, \ldots, x_n) \\
t_3 &= max(\{x : x \leq Q_{75\%}(x_1, \ldots, x_n) + 1.5 * IRQ(x_1, \ldots, x_n)\})
\end{aligned}
\tag{4.1}
$$

The QATCH approach for utility functions uses these three bounds to define not just a minimum and maximum range, but also a middle bound. Linear interpolation is applied to accomplish a stepwise-linear utility function that is aware of the true median value. The difference between one-step and two-step linear interpolation approach As shown in figure 4.10, a one-step utility function (left graph) is simply a linear connection between worst and best range. A two-step utility function (right graph) provides more precision by introducing a median target as well. As a default, PIQUE follows the approach used by QATCH of two-step linear interpolation.

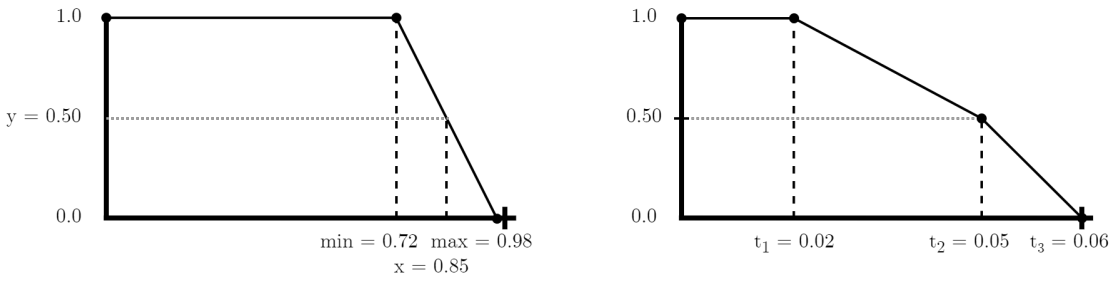

Figure 4.10: Comparison of two utility function strategies.

For example, let the two-step linear interpolation utility function from figure 4.10

represent the result from benchmarked thresholds of a measure named *Encryption Smells*. During project assessment, if the project's measure value of *Encryption Smells* was 0.05, the utility function of *Encryption Smells* will transform its value into $measureEval(EncryptionSmells) = 0.50$.

Unlike QATCH, PIQUE applies utility functions at the measure level instead of the product factor[8] level. Utility functions can only be applied at the product factor level if it can be guaranteed that only one layer of product factors exists and each product factor only has one measure attached to it. Because PIQUE aims to facilitate experimental model designs, this guarantee cannot be assumed.

Subjective Factor Weighting PIQUE's default factor weighting follows the AHP strategy [34] as shown in QATCH [37] to introduce quality subjectivity with one difference: instead of using numerical ratio values for pairwise comparison, the linguistic values ["very low", "somewhat low", "equal", "somewhat high", "very high"] are used.

First, hand-entered values are obtained from the stakeholder of the product under evaluation as a collection of files called comparison matrices. The comparison matrices are a collection of pairwise comparisons at each level of the quality model factor hierarchy. Assuming the default PIQUE quality model structure is being used,[9] comparisons will only occur at the single quality aspect layer and the single product factor layer.

For example, if the example quality model description of Figure 4.2 is being used, four comparison matrices will need to be filled. One matrix is for pairwise comparisons of the quality aspects in the context of total software quality. The other

[8]Referred to as the properties level by QATCH.

[9]The structure presented by QATCH: one layer of quality aspects and one fully connected layer of product factors.

Table 4.3: Comparison matrix for the quality aspect layer

TQI	Portability	Maintainability	Security
Portability	-	very low	low
Maintainability	-	-	somewhat high
Security	-	-	-

Table 4.4: Comparison matrix for the product factors in context of portability

Portability	Modularity	Redundancy	Documentation	Encryption
Modularity	-	very high	somewhat high	very high
Redundancy	-	-	somewhat low	equal
Documentation	-	-	-	somewhat high
Encryption	-	-	-	-

three matrices are for pairwise comparisons of the product factors in the context of the three quality aspects: (1) the product factors in the context of portability, (2) the product factors in the context of maintainability, and (3) the product factors in the context of security. The matrices are diagonal, so diagonal entries (e.g. Portability:Portability) have implied values of "equal", and entries across the diagonal have implied values of their reciprocal (e.g. if Portability:Maintainability is "very low", then Maintainability:Portability is "very high"). An example of the quality aspects matrix in the context of total software quality is shown in table 4.3 and the first product factors matrix (in the context of portability) is shown in table 4.4.

Using the comparison matrix values of tables 4.3 and 4.4, PIQUE will translate the linguistic values in to [{very low = 0.1111}, {somewhat low = 0.25}, {equal =

Table 4.5: Quality aspects → TQI derived weights

	Portability	Maintainability	Security
TQI	0.0658	0.7171	0.2172

Table 4.6: Product factors → Portability derived weights

	Modularity	Redundancy	Documentation	Encryption
Portability	0.6568	0.0625	0.2183	0.0625

1.0}, {somewhat high = 4}, {very high = 9}].[10] After running the AHP procedure, the derived weights for the quality aspect to TQI layer are given in table 4.5 and the product factors weights in the context of portability are given in table 4.6.

Model Node Evaluation Functions PIQUE provides the utility to override how any nodes in the quality model evaluates their assessment value, but it is valuable to provide a default method of node evaluation. These default functions differ depending which layer of the model is being evaluated.

A diagnostic node's default evaluation strategy is the sum of its finding values. For example, in the context of a code smell diagnostic, if the tool associated with diagnostic *SCS0005* found 17 occurrences of *SCS0005* in the product under assessment, diagnostic *SCS0005* would evaluate to 17; that is, 17 child `Finding` object instances, all with value *1*. In the context of a metric, a metric diagnostic *Lines of Code* will evaluate to the result of the LoC tool's finding value, a field whose value represents the lines of code; that is, 1 `Finding` object with a value *30,000* if the system has 30,000 lines of code. Depending how a tool outputs its finding values, there are

[10]These values come from table 1 of [34, p. 86] for the intensities 1, 4, 9 and their reciprocals.

many situations where this default behavior should be overridden.

Measure nodes evaluate by applying their utility function to the normalized value of the sum of their diagnostics.

$$valueOf(measure) = utilityFunction(normalize(\sum_i valueOf(diagnostic_i))) \tag{4.2}$$

If a measure node is evaluated by a function different than the sum of its children–such as the coupling measure described in section 2.2–the default node evaluation function will need to be overridden.

All quality model factor nodes (TQI, quality aspects, and product factors) evaluate by the weighted sums of their children by default. If a bottom level product factor node has more than one measure child and the incoming measure edges do not have weights assigned, the evaluation assumes equal weighting.

4.3 Overriding Mechanisms

Modification of the derivation and assessment mechanisms is a key feature that allows PIQUE to be more than just a "quality assessment on rails" framework. By providing a way to modify internal mechanisms while still easily using the platform for quality control activities, PIQUE can act as a true platform for investigative quality research while removing the large overhead currently restricting the field.

In general, overriding the following mechanisms uses the same approach: in the researcher's PIQUE-EXT system, instance a class that implements the interface of the desired mechanism and update the model configuration if needed. UML for the classes mentioned in this section is provided in section 4.1.

Normalization Functions

Normalization at the measure layer is customizable through two steps. First, in PIQUE-EXT, a normalizer class that implements the `INormalizer` interface should be created. The interface requires an implementation of the `normalize()` function given a collection of `ModelNode` objects as parameters. Second, *norm_strategy* in the quality model description configuration file should be defined according to the name of the normalizer class object for the desired node to apply the normalization strategy to.

Utility Functions

A model node's utility function can be overridden by implementing one or more `IUtilityFunction` classes in PIQUE-EXT . Defining the `utilityStrategy()` method involves writing a function that modifies the model node's output in some way given the model node's state. Typically this will be applied to measure nodes using the threshold data that comes from the benchmark repository. The new utility function class can then be assigned to model node entries in the quality model description configuration file using the utility function's full class name as defined in PIQUE-EXT.[11]

Model Node Evaluation Functions

All quality model nodes in the framework extend the abstract `ModelNode` class which requires an abstract `evaluate(IEvaluator)` method. Two approaches in node evaluation strategy are supported. If an entire collection of node types should be modified in the same way–for example all product factors should be multiplied by some scalar–the product factor *.json* object in the quality model description file can have a child field named *eval_strategy* listed with the `IEvaluator`'s full class name as

[11]For example, `myproject.pique.ruby.CustomUtility`.

the value. If a specific node requires evaluate modification, the same approach can be used, but at the specific node's *.json* object entry.

Subjective Factor Weighting

The framework assumes that manually entered subjective quality expressions of the factor hierarchy are given from an external source (such as *.json* input). By default, the `DefaultWeighter` expects well-formed AHP matrices as described in section 4.2.

For modification, the syntax of the text-format input does not matter as long as the modified approach defines how to translate the input into the relevant factor edges of the quality model. To accomplish this, the `IWeighter` class can be implemented in PIQUE-EXT. The framework looks for the `WeightResult` objects produced by the `IWeighter` interface and automatically applies the results to the quality model object. Similar to benchmarking modification, the new weighter class can be passed to the RUNNER module which is automatically detected and used by the framework instead of the default weighting approach.

Benchmark Methodology

While the framework always expects a benchmark repository, the benchmarking procedure can be modified. The `DefaultBenchmarker` class (run by PIQUE during model derivation) returns an array of the min, median, and max value of each measure found in the benchmark repository after removing outliers, but this could be modified to, for example, not remove outliers, give more weight to specific benchmark projects, or utilize machine learning techniques.

To override, a `Benchmarker` class should be created in PIQUE-EXT implementing `IBenchmarker`. Implementing the benchmark class requires defining a method that takes as input the path of the benchmark repository and the `ITool` instances needed

to run the static analysis tools on the projects, and it returns a mapping of model nodes and decimal arrays. The new benchmark class can then be passed in to the RUNNER module where PIQUE will automatically detect to use the new strategy.

4.4 Model Derivation Process

Before quality assessment can occur on a product, a quality model is needed. Model derivation is the process by which a valid model can be generated. This section assumes the necessary inputs[12] are already provided and focuses on the derivation process itself as implemented in the PIQUE calibration component. Figure 4.11 shows this process in visual form.

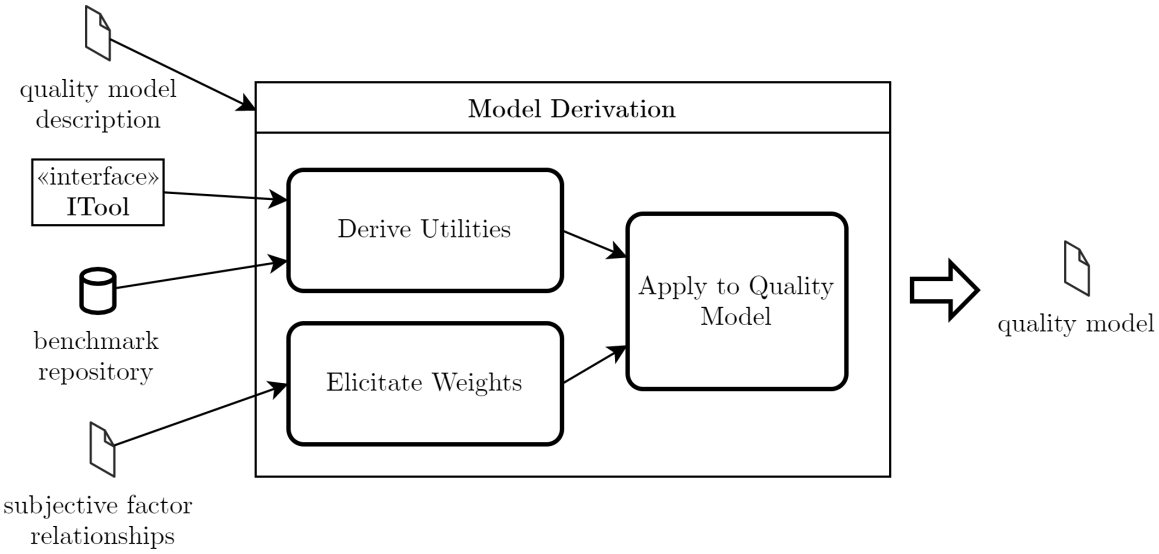

Figure 4.11: Model derivation process

Utility Functions First, the function `IBenchmarker.deriveThresholds()` is called using the benchmark repository, quality model description configuration file, and

[12]Quality model description *.json* file, `ITool` classes, subjective factor importance descriptions, and the benchmark repository directory.

`ITool` objects as input. The platform iteratively steps through each project in the benchmark repository folder, audits the project using the given `ITool` objects, collects the tool findings as `Diagnostic` objects, and uses the diagnostics to evaluate the measure layer (with normalization applied) for the given project.

If the benchmark repository is empty or no valid files are found in the directory, the platform alerts the user with an error and exits the derivation process. For initial testing purposes, a user could provide a benchmark repository consisting of just the project they intend to assess, given building a benchmark repository can be a time-consuming process. The collection of all normalized measure nodes across all benchmark projects is then used to return a collection of `Map<String, Double[]>` objects representing the measure names and their associated utility thresholds.

Subjective Weight Elicitation Next, `IWeighter.elicitateWeights()` is called using the path to the subjective factor weighting files (by default, *.json* comparison matrices) as input. The weighting strategy as defined by `elicitateWeights()` returns a collection of `WeightResult` objects that maps each factor to its incoming edge weight values.

Model Generation The utility thresholds to each measure node and the weights for every factor → factor relationship are applied. The platform finally returns a fully derived quality model (see Figure 4.3) as both a JVM object and in *.json* file format.

4.5 Product Assessment Process

Product assessment is an act of quality control accomplished by using a language-specific quality model on a target product resulting in a model with values representing each node. Figure 4.12 shows a visual representation of PIQUE's

approach to assessment.

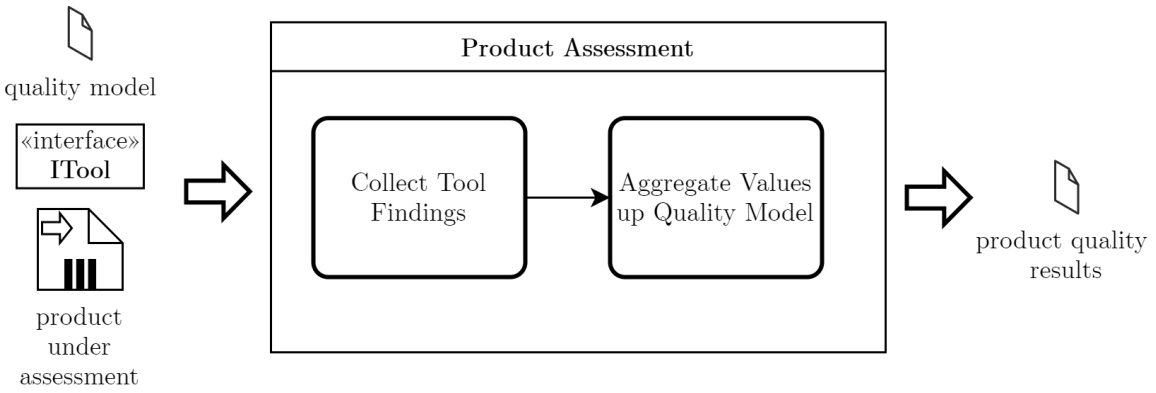

Figure 4.12: Model assessment process

As input, the RUNNER component expects a derived quality model, the necessary language-specific `ITool` objects for static analysis, and the path to the product under assessment. Next, the `ProjectEvaluator::evaluateProject()` function activates the `ITool` objects resulting in a collection of `Finding` and `Diagnostic` objects that are applied to the quality model. The measure layer of nodes are evaluated using the defined utility function and normalization strategies as provided by the researcher (or by default). Finally, the factor layer of nodes (product factors, quality aspects, and total quality index) are evaluated, by default using aggregation of weighted sums.

The model at this point represents an assessed product with a form similar to Figure 4.13. A *.json* representation of the model is output with decimal values accessible at every node, ready for use by continuous integration and desired visualization techniques.

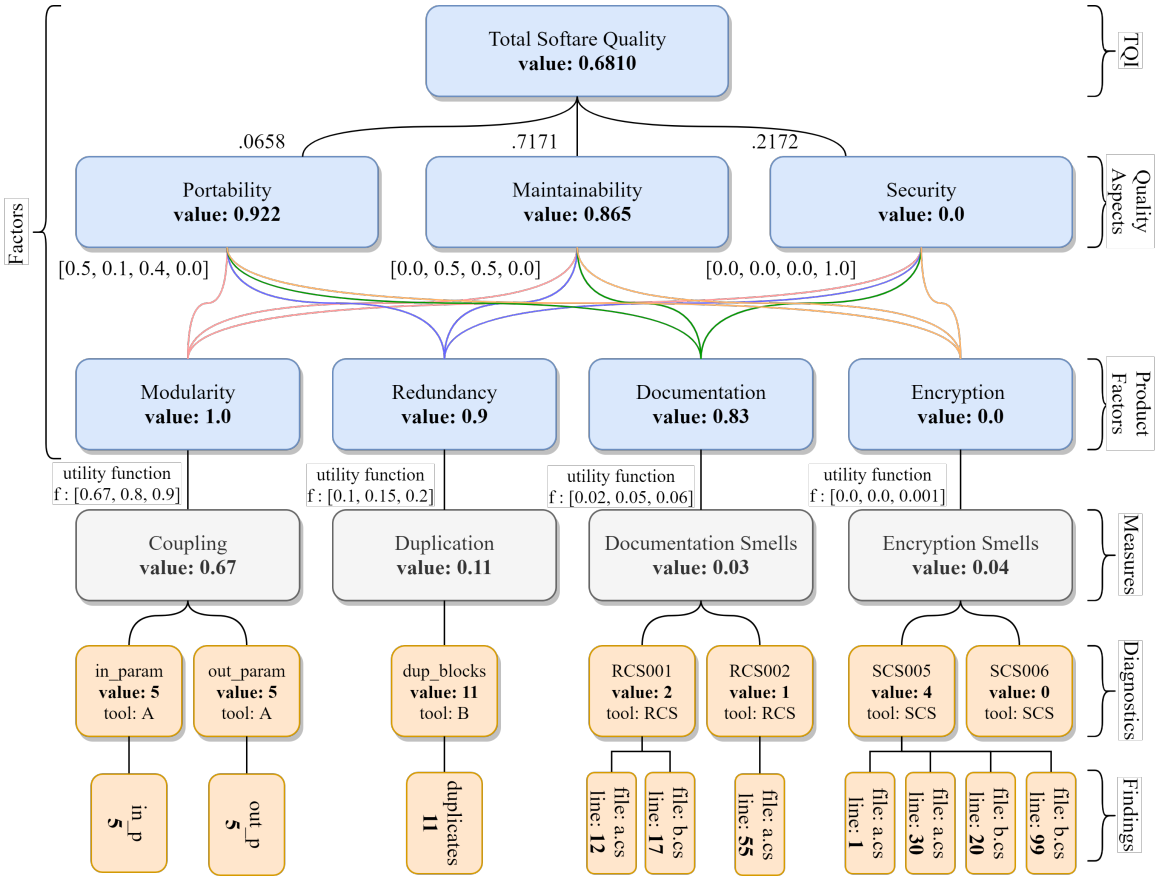

Figure 4.13: A model with values representing a product's assessment. Practical models will often have thousands of diagnostics.

CHAPTER FIVE

PIQUE OPERATIONALIZED: NEW MODELS; NEW TOOLS

5.1 Introduction

The purpose of the platform presented in chapter 4 is to create the foundation necessary to obtain metrics and answer the questions presented by the GQM[1] of section 1.2. However, the platform itself is a collection of language-agnostic API calls and libraries. To instantiate a quality model and assess actual projects, a language-specific extension of PIQUE must be created and used for evaluation.

This chapter describes the process necessary to derive a new model and use it for product assessment in both academic and industrial settings. This involves describing a new quality model, finding static analysis tools, connecting the tools using PIQUE API points, building a benchmark repository, filling in comparison matrices to introduce subjective quality opinion, and running the main derivation method using PIQUE. The derived model can then be used to assess quality of real systems in a specific programming language.

Two models are constructed in this chapter. This first is a fully operationalized ISO/IEC 25010 based [14] total quality model runnable on C# systems. The second is an experimental security model based on the hierarchies presented by ISO/IEC 25010 and MITRE's Common Weakness Enumeration[2] (CWE), also intended for C# systems. The total quality model instantiation is used to generate metrics for the *in vivo* and *in vitro* test cases of chapter 6.

[1] Goal/Question/Metric paradigm.
[2] `https://cwe.mitre.org/`

5.2 Deriving and Using a C# Quality Model

The total quality model derived and used in this chapter addresses C# systems: specifically C# systems on .NET Standard 2.0. The reason for this choice in language support is three-fold.

(1) Nearly all historic quality modeling attempts have targeted Java. This is primarily due to the robust and open-source tool support available to use in Java systems. Of the quality modeling approaches presented in chapters 2 and 3, one attempt provided C# operationalization [40]. However, the operationalization used only one static analysis tool providing 146 rules combined with a small benchmark repository of 23 systems. As a result, the authors rejected their C# operation for empirical validation. Additionally, C# systems can be notably unfriendly to researchers due to .NET integration with Visual Studio and lack of open-source third party involvement. Thus, in order to answer GQM questions relating to the improvement of experimentation capability and time investment relating to quality model processes, designing a C# quality model is a ripe domain to run test cases.

(2) For *in vivo* validation, exercises are run with systems from two companies in industry. The two systems are written in .NET, thus, design of C# quality assessment is the sensible and necessary language to target.

(3) In United States government agencies and contractors, .NET is frequently used as the framework of choice due to Microsoft's security features and product integration with government agreements. Because funding for research in the domain of software analysis is often driven by Department of Defense support, the ability to apply research products to pre-existing government or contractor systems is valuable.

A C# Quality Model Description

The first step when deriving a new model is writing a quality model description. Discussed with technical details in section 4.1, the quality model description is a *.json* text file outlining the desired model node names, the factor hierarchy, the measure layer, and the diagnostics provided by tools necessary to evaluate the measures. The model description used in this chapter uses the ISO/IEC 25010 hierarchy [14] while maintaining one layer of quality aspects, one layer of product factors, and a single measure for each product factor as recommended by [37].

Factors The factor hierarchy described by the quality model description is depicted in Figure 5.1. The approach used to architect this model is to build the factor hierarchy first and then use a bottom-up approach by evaluating the available tools, their diagnostics, and how those diagnostics can connect to measures.

A design choice is made to construct the model using only measures of findings, no metrics.[3] This design decision expresses the opportunity of using PIQUE to create experimental models. The quality models reviewed in chapters 2 and 3 primarily use metrics at the bottom layer of their hierarchies. Instead, this model provides insights into whether a model that evaluates using only data from findings still produces valid results.

Of particular interest is the selection of product factors shown in Figure 5.1. A product factor is defined as factor nodes that can decompose into directly measurable concepts that represent the attributes of the part of the product (section 4.1). Due to the generality of a product factor's definition, how should one decide how many product factors to include?

A lesson learned while building this model is that a model is only as strong

[3] Apart from the metrics needed for normalization such as LoC.

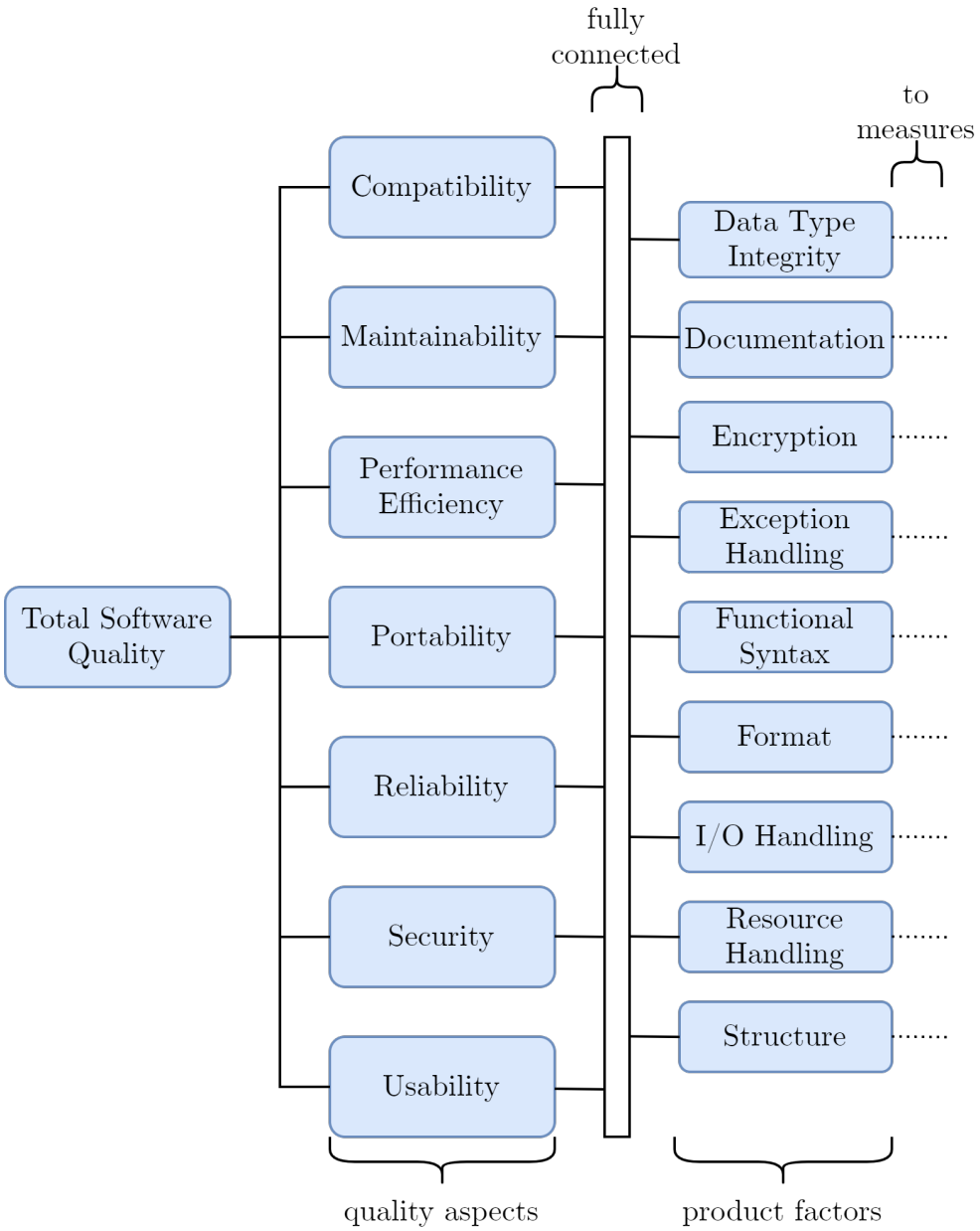

Figure 5.1: C# quality model description factors

as the tools that support it. While a menagerie of product factor nodes could be contrived in one's head, introducing every possible product factor into the model is pointless if there is no reasonable way to measure it given the available tools. Instead, the approach given here uses a bottom-up approach by using assessment of

the various product factors the tool diagnostics could represent to drive the design of the measure layer of nodes and the bottom-level product factors. Ten categories of low-level measurable concepts result from this assessment with definitions shown in appendix G.

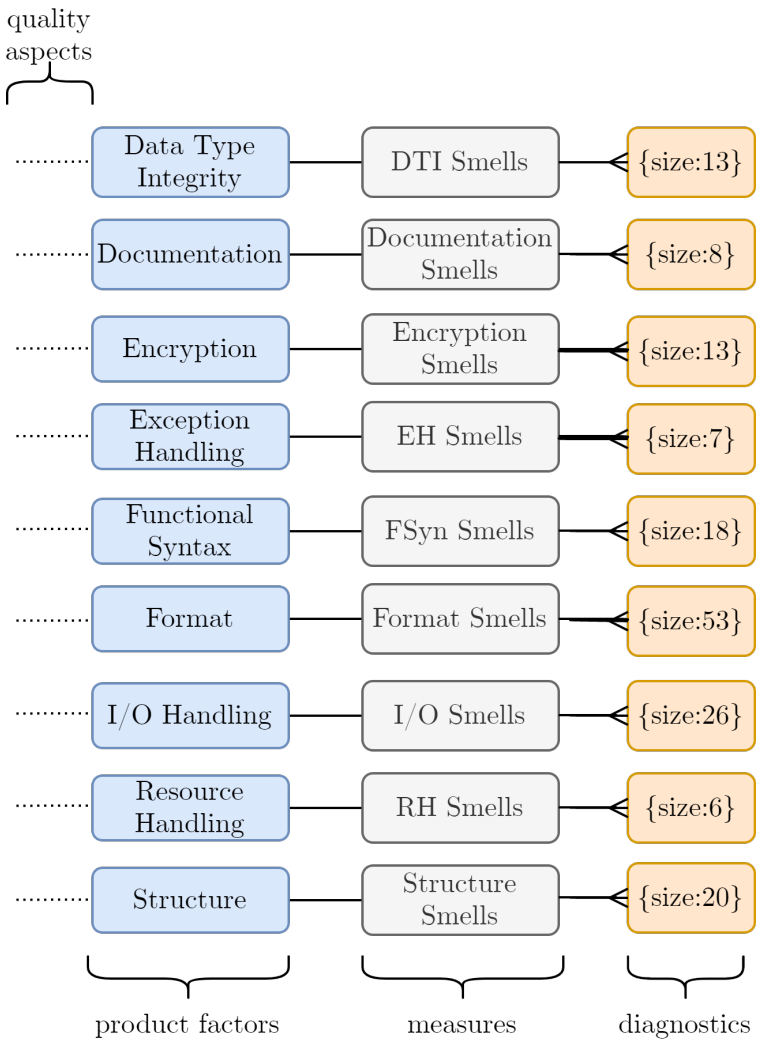

Figure 5.2: C# quality model description: measures and diagnostics

Measures and Diagnostics Figure 5.2 shows the bottom half of the model represented by the C# quality model description file. Because the experimental approach of this model is using only findings of static analysis tools, and each finding

provided by the tools represents a negative element in the system, every measure is labeled as a smell. Thus, the heuristic represented by this quality model is that all product factors start with a perfect value of 1.0, but finding problematic code–expressed as code smells–subtracts from the perfect values. This is how product factors can be measured by the existence or non-existence of smells which are categorized as a collection of diagnostics.

Integrating Static Analysis Tools

The selection and integration of static analysis tools and quality model design at the measure and diagnostic layer go hand-in-hand, so the two steps typically happen simultaneously. To integrate, a new project designed for C# analysis is made, PIQUE-CSHARP, which uses PIQUE as a dependency.

For this project, a third party tool named Roslynator[4] is used which provides command line interface utilities to run static analysis on .NET systems. Along with the hundreds of diagnostics provided by default with Roslynator, the tool Security Code Scan[5] is added for additional support of security-focused findings and the Microsoft.VisualStudio.Threading xplat library[6] is added for detection of asynchronous execution code smells. Roslynator allows all of these tools to run simultaneously through its command line interface. A technical document showing the specific steps taken to integrate Roslynator, Security Code Scan, and VS-THREADING is provided in appendix C.

To connect the Roslynator CLI to PIQUE, a class in made in the PIQUE-CSHARP system named `Roslynator` which implements the `pique.analysis.ITool` interface. After implementing the required methods, there is now an `ITool` class that

[4] `https://github.com/JosefPihrt/Roslynator`
[5] `https://security-code-scan.github.io`
[6] `https://github.com/Microsoft/vs-threading`

knows how to run Roslynator analysis on a C# system and interpret the results into `Diagnostic` JVM objects.

Building a Benchmark Repository

The benchmark repository is a collection of open source C# systems used to calibrate the measure utility functions. Details on these systems, including their GitHub URL, lines of code, logical lines of code, and comment lines are given in appendix D.

The repository used in this operationalization was personally constructed for PIQUE-CSHARP, contains 45 C# systems, and represents a cumulative 3.4 million lines of code. These systems were gathered by searching `https://github.com`, filtering by the C# language, and sorting by most stars. In order to analyze using the static analysis tools discussed previously, the C# systems are verified to compile using .NET Framework 4.6.x. Finally, minor cleaning of the projects occurred by ensuring the static analysis tools were able to find the main entry point[7] and irrelevant side code is removed.

Filling in Comparison Matrices

The approach for utility function derivation used in this operationalization of PIQUE follows the analytical hierarchy process approach presented by QATCH [37]; however, PIQUE features the ability to enter linguistic values instead of numeric ratio values. Given there is one layer of seven quality aspects, and one layer of product factors, eight comparison matrix files–one for the total quality node and seven for the project factors–in *.csv* format are created and ready for hand entry.

To introduce industry subjectivity, a local company filled in the AHP values

[7] A *.sln* file at the project root directory level.

of the quality aspect → TQI layer according to the aspects of software quality they subjectively valued. The filled in matrix[8] for the top layer of the factor hierarchy is shown in table 5.1. The full collection of comparison matrices are given in appendix E.

Table 5.1: TQI Comparison matrix from practitioner interaction

TQI	Comp.	Maint.	Perf.	Port.	Rel.	Sec.	Use.
Comp.	-	SL	SL	SL	VL	VH	EQ
Maint.	-	-	EQ	SH	SL	VH	SH
Perf.	-	-	-	SH	SL	VH	EQ
Port.	-	-	-	-	VL	VH	EQ
Rel.	-	-	-	-	-	VH	SH
Sec.	-	-	-	-	-	-	VL
Use.	-	-	-	-	-	-	-

Running Derivation and Assessment

Model Derivation With a quality model description filled in, static analysis tools connected as `ITool` classes, and comparison matrices and the benchmark repository prepared as input, a model can now be derived. Within PIQUE-CSHARP, the platform method `pique.runnable.QualityModelDriver.deriveModel()` is called. This returns a derived C# quality model–now with edge weights and utility functions–in *.json* format. Tables of the derived weights can be found in appendix F with the full quality model, including utility thresholds, provided in appendix G.

[8]VL: very low. SL: somewhat low. EQ: equal. SH: somewhat high. VH: very high

Project Quality Assessment The final step of quality model operationalization is assessment. Passing in the the same `ITool` classes used in model derivation as arguments, the *.json* derived quality model, and the path to the C# system under assessment, the platform method `pique.runnable.Evaluator.runAssessment()` is called. The method returns a *.json* file representing the assessment results. The result file follows the structure of the quality model file used as input, but the quality model now has findings instanced at the bottom layer and numerical evaluation values for all other nodes.

It is these assessment result files along with metrics produced during quality model design phases that are used to drive the exercises shown in chapter 6 to validate the system.

5.3 A C# Security Model

Improvement of quality model experimental opportunity is a component of goal G01 from section 1.2. To present a case in support of the experimental opportunity PIQUE provides, this section shows how the platform can be used to design a security quality model for C#. Security, in the context of static quality analysis, is one subcomponent described by ISO/IEC 25010. Of the historical models reviewed in chapters 2 and 3, no model specifically focuses on security, and those that do include security as a quality aspect node–for example, QATCH [37] and Quamoco [40]–calculate their security node values through relatively weak approaches.

QATCH evaluates security through the weighted sums of 11 factors–*Coupling*, *Redundancy*, *Bad Function*, *Structure*, *Assignment*, *Resource Handling*, *Cohesion*, *Comprehensibility*, *Complexity*, *Messaging*, and *Encapsulation*–of which only *Encapsulation* and *Resource Handling* have significant weighting (0.3382 and 0.1811 respectively). The primary influence on the security score, *Encapsulation*, is simply

the value of the encapsulation property of the system. Security is a complex concept, so evaluating its quality primarily off of a single property is likely unsatisfying to a practitioner.

Quamoco's object-oriented base models refine security using six factors: *Interface Permission Consistency*, *Runtime Environment Independency*, *Untrusted Data Sanitization*, *Definition and Usage Consistency Regarding Scope*, *Runtime Environment Independence*, and *Encapsulation Strength*.

Of these six factors, *Interface Permission Consistency* has 1 measure, *Runtime Environment Independency* has 2 measures, *Untrusted Data Sanitization* has 7 measures, *Definition and Usage Consistency Regarding Scope* has 1 measure, *Runtime Environment Independence* has 1 measure, and *Encapsulation Strength* has 16 measures. In similar fashion to QATCH, six product factors with a few associated measures is not enough granularity to represent the security domain in a satisfying way.[9]

This lack of granularity drives the motivation to present a security-focused quality model. Demand for security quality assessment is increasing significantly by both private and government entities due to the impacts security flaws can have on elections, national security, public trust, stocks, and lawsuits.

During work on this section, it was realized that there is not enough non-proprietary C# static analysis tools capable of identifying security vulnerabilities to operationalize a satisfying C# security quality model. Instead, this section constructs the quality aspects and product factors of a security quality model according to PIQUE constructs, but leaves as future work the implementation of other layers necessary to actualize the model. The purpose of this section is to present the

[9]To the credit of the Quamoco team, the intent of the base model provided is for further extension rather than presenting a fully realized model; however, a Quamoco model with security additions was not available.

initial work needed to construct a full security quality model, assist future research in recognizing the amount of work necessary to operationalize such a model, and help guide strategies and priorities to approach the challenge.

Model Overview

The security model presented in this section is constructed using three primary concepts. (1) Security is the top node of the quality model. (2) The quality aspects come from the ISO/IEC 25010 definition of security quality. (3) The low-level quality aspects and product factors come from MITRE's community developed Common Weakness Enumeration (CWE).

Common Weakness Enumeration The Common Weakness Enumeration [36] is a categorization and structuring of software weaknesses and vulnerabilities. It is a community driven project, sponsored by the United States National Cybersecurity FFRDC, which gives notable effort toward static analysis tool support. The CWE has encouraging potential for exploratory security quality assessment opportunity by also linking to the Common Vulnerabilities and Exposures[10] (CVE) catalog, a collection of publicly known information-security vulnerabilities and exposures. The CVE has an associated open industry standard for security vulnerability severity assessment called the Common Vulnerability Scoring System (CVSS). The standardizations, community support, and interconnection of CWE, CVE, and CVSS makes the CWE a prime structure to mirror for a security quality model.

The CWE is large, boasting over 600 categories and over 1000 specific weaknesses. To categorize and organize such a large collection, the CWE offers different views of the weaknesses–for example, from the view of web applications,

[10] `https://cve.mitre.org/`

hardware design, software development, and architectural concepts. Given the goals of this thesis focus on the view of static software analysis, the CWE architectural concepts view (CWE-1008) and the CWE research concepts view (CWE-1000) are most relevant. The architectural view is chosen to guide the initial quality aspects under the ISO/IEC 25010 quality aspects, then the natural hierarchy the associated architectural concepts and research concepts weaknesses formed by the CWE are used to build the product factor tree.

Model Design

Quality Aspects The model is designed from a top-down approach. Security is set as the root node. Next, the ISO/IEC 25010 [14] security sub-characteristics[11] are used to construct the top layer of quality aspects. The next layer of quality aspects uses the 12 categories of the CWE architectural concepts: *Auditability*, *Actor Authentication*, *Actor Authorization*, *Cross Cuttability*, *Data Encryption*, *Actor Identification*, *Access Limiting*, *Exposure Limiting*, *Computer Locking*, *User Session Management*, *Input Validation*, and *Message Integrity Verification*.[12] In the quality model design, the nodes for the layers presented so far are labeled as quality aspects due to their expression of abstract quality goals that cannot be measured directly.

For the next layer of factors, the CWE architectural concepts category children are introduced. The CWE at this level defines entries as one of four types: a pillar node, a class node, a base node, and a variant node.

A pillar node is defined as, "The highest-level weakness that cannot be made any more abstract. Pillars are the top-level entries in the Research Concepts View (CWE-1000) and represent an abstract theme for all class/base/variant weaknesses

[11] Confidentiality, Integrity, Non-repudiation, Authenticity, and Accountability

[12] These names are slightly modified from how they appear in the CWE to have an action-verb semantic.

related to it. A pillar is different from a category as a pillar is still technically a type of weakness that describes a mistake, while a category represents a common characteristic used to group related things."

A class node is, "A weakness that is described in a very abstract fashion, typically independent of any specific language or technology. More specific than a Pillar Weakness, but more general than a Base Weakness. Class level weaknesses typically describe issues in terms of one or two of the following dimensions: behavior, property, and resource."

A base node is defined as, "A weakness that is described in an abstract fashion, but with sufficient details to infer specific methods for detection and prevention. More general than a Variant weakness, but more specific than a Class weakness."

Finally, a variant node is, "A weakness that is linked to a certain type of product, typically involving a specific language or technology, more specific than a Base weakness. Variant level weaknesses typically describe issues in terms of 3 to 5 of the following dimensions: behavior, property, technology, language, and resource".

Table 5.2: CWE categories to factor mappings

CWE Category	*Factor Type*
Pillar	High-level product factor
Class	High-level product factor
Base	Low-level or leaf product factor
Variant	Low-level or leaf product factor

In general, nodes of type pillar, and class cannot be measured directly, so they fit naturally as high-level, non-leaf node product factors that decompose into their CWE base and variant children. Nodes of type base and variant generally have some way

to be directly detected or measured, thus they naturally fit as bottom layer product factor nodes.

The connection of ISO/IEC 25010 quality aspects to CWE product factors completes the factor hierarchy. Security is the root node. The five ISO/IEC security subcharacteristics are the next layer of quality aspects, and the 12 CWE architectural concepts are chosen to be the final layer of quality aspects. In total, this represents 18 quality aspect factor nodes.

Product Factors The product factors are a massive hierarchy that mirror the child CWEs of each architectural concept node as provided by the Common Weakness Enumeration,[13] but CWEs that cannot be detected by C# static analysis tools are removed.[14] Some of the product factor hierarchies have deep sub-trees up to eight layers deep, such as child nodes of the *Actor Authorization* or *Input Validation* quality aspects. This mirroring of hierarchies is chosen due to the benefit of CWE structure having strong community agreement on representing a sensible decomposition of security concepts.

Figure 5.3 presents a high-level view of the quality aspect and product factor design of the security quality model. In total, there are 18 quality aspects and 984 product factors. Of the product factors, 85 are pillar or class types, and 899 are base or variant types. Because nearly all base or variant type CWEs can be directly measured by C# static analysis tools, nearly 899 diagnostics with real static analysis tool support must be available in order for this model to be operationalized.

[13] `https://cwe.mitre.org/data/definitions/1008.html`

[14] Some CWEs are language specific such as Java-focused CWE-382: "J2EE Bad Practices: Use of System.exit()".

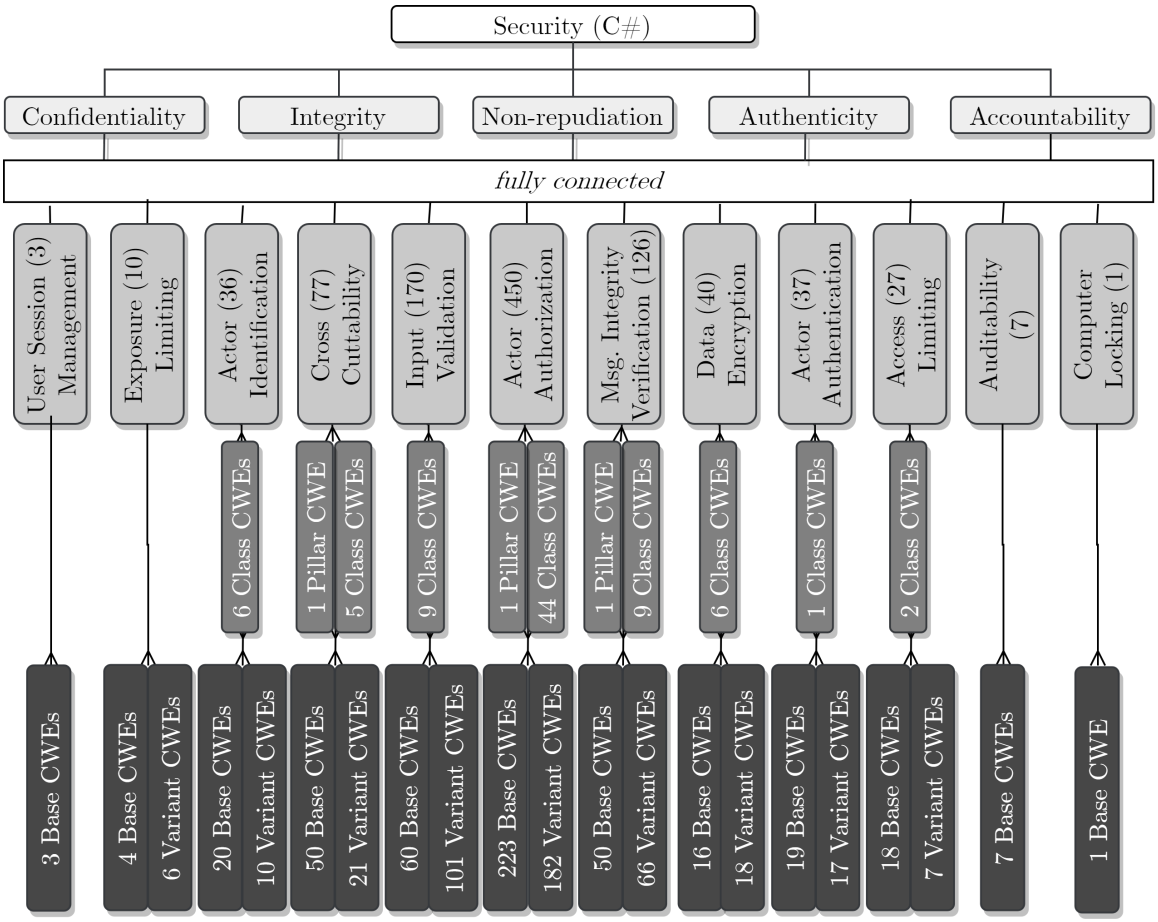

Figure 5.3: High-level view of the factors for a C# security quality model.

Examples Figure 5.4 presents an example of quality aspect *Input Validation*'s decomposition into diagnostic *SCS0002 - SQL Injection (LINQ)*. SCS0002 is a C# diagnostic supported by the tool *Security Code Scan*[15] which triggers when code similar to listing 1 is found in a system.

Figures 5.3 and 5.4 are presented to give an intuitive notion of how large a fully supported security model can be. The presented *Input Validation* quality aspect node of Figure 5.4 is 1 of 12 CWE architectural concept nodes. In the case of *Input Validation*, its product factor tree contains 170 child nodes with approximately[16]

[15]https://security-code-scan.github.io/

[16]Not all base or variant nodes are leaf nodes.

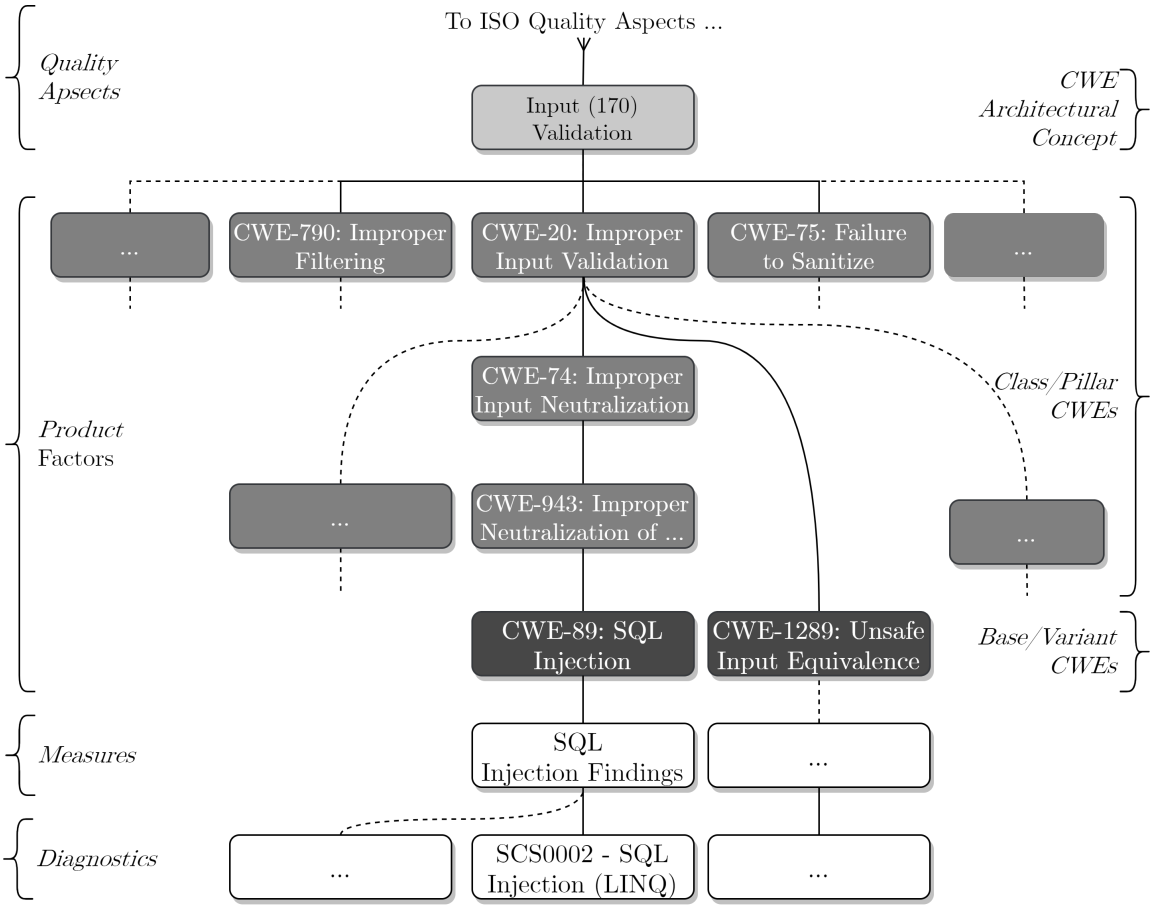

Figure 5.4: Path from CWE quality aspect to C# tool-supported diagnostic.

```
db.ExecuteQuery(@"SELECT name FROM dbo.Users WHERE UserId =
    " + inputId + " AND group = 5");
var query = "SELECT name FROM dbo.Users WHERE UserId =
    " + userId + " AND group = 5";
var id = context.ExecuteQuery<IEnumerable<string>>(query)
    .SingleOrDefault();
```

Listing 1: Security Code Scan *SCS0002*

167 product factor leaf nodes with associated measures requiring C# specific static analysis tool support. The path in Figure 5.4 from quality aspect *Input Validation* to diagnostic *SCS0002 - SQL Injection (LINQ)* gives a sense of what the 100s of subtrees may look like.

Regarding weighting, the AHP process used in section 5.2 is infeasible due to the deep and wide design of this quality model: the pairwise comparisons would likely take months to enter. The topic of the best weighting strategy to use for a model such as this is outside of the scope of this work, but the componentized design of PIQUE supports whatever approach is used through use of the factor weighting interface calls discussed in section 4.2. A potential approach could be to use AHP[17] for the quality aspect layer edge weights and reactively set the CWE product factor edges in accordance with vulnerability threat levels from live-feed resources such as the Common Vulnerability Enumeration.[18] This area of dynamic weighting has good potential for machine learning applications as well.

5.4 Conclusion

Two PIQUE models are presented in this chapter: A small, but fully operationalized C# total quality model and a large, conceptual C# security model. Both models exemplify how the concepts and constructs of PIQUE can facilitate notably different model designs and strategies. The application of PIQUE model design into security quality assessment is of particular interest due to the lack of research in the area and demand by civilian and government entities. By presenting the full process of operationalizing a simple model and presenting a complex security model design, the initial pieces are in place for further research.

[17] Analytical Hierarchy Process. Discussed in section 4.2.
[18] `https://cve.mitre.org/`

CHAPTER SIX

TEST CASES

6.1 Introduction

This chapter presents a collection of test cases used to generate data to verify the capabilities of PIQUE and help answer the questions presented in section 1.2. These exercises utilize the platform described in chapter 4 and the operationalized C# model generated by PIQUE described in chapter 5. The metric data resulting from these test cases are used to drive the discussions and analysis of chapter 7. As a reminder, the goals, questions, and metrics are as follows:

G01: Analyze a process of generating, validating, and operationalizing quality models for the purpose of improvement with respect to effort investment, experimentation, and collaborative opportunity from the point of view of quality model researchers in the context of static software system analysis.

Q01-01: How much effort does it take to generate a model using default mechanisms?

Q01-02: How much effort does it take to generate a model using modified mechanisms?

Q01-03: How much effort does it take to operationalize a model?

Q01-04: Do the models produced facilitate ease of evaluation by researchers?

Q01-05: Are the models produced valid?

G02: Analyze a process of generating and operationalizing quality models for the purpose of improvement with respect to cost investment and acceptability from the point of view of software development practitioners in the context of static software system analysis.

Q02-01: Is it expensive to tune a model to a company's needs?

Q02-02: How difficult is it to integrate quality model assessment into an external, continuous integration system?

Q02-03: Can quality models be used such that their output values are trusted by practitioners?

Table 6.1: Effort Metrics

M01:	Man-hours taken to install PIQUE as a library resource.
M02:	Man-hours taken to connect static analysis tools to PIQUE.
M03:	Man-hours taken to design a quality model description *.json* file.
M04:	Man-hours taken to prepare a benchmark repository.
M05:	Man-hours taken to modify a default normalization function.
M06:	Man-hours taken to modify a default utility function.
M07:	Man-hours taken to modify a default weighting function.
M08:	Man-hours taken to modify a default evaluation function.
M09:	Man-hours taken to operationalize a derived quality model.
M10:	Time taken to run model derivation (benchmark repo size = 44).
M11:	Time taken to run system quality assessment.
M12:	Time taken by a practitioner to express subjective quality definitions.

Table 6.2: Quality Assessment Metrics (boolean values)

M13:	The change in quality assessment score decreases after introducing flaws.
M14:	The change in quality assessment score increases after introducing improvements.
M15:	The change in quality assessment score reflects modified subjective quality opinion changes.

Table 6.3: Design Metrics

M16:	Model output exposure.
M17:	Number of external dependencies needed for actualization.
M18:	[True | False] Failure of assessment does not interfere with other system processes

Table 6.4: Subjective Metrics (boolean values)

M19:	The slope of assessed values over time matches practitioner opinion.

Table 6.5: Requirement Metrics (boolean values)

M20:	The platform is open source.

6.2 Test Designs

The test cases fall in to one of three general categories: model construction/implementation effort, model understandability, and model assessment result validity. Model construction and implementation tests involve tracking effort data for the time and resources needed to design a model from scratch and run its

operationalized assessment on real software products. Model understandability tests involve generating data about the output files of assessment results to verify assessment component exposure and if the data expressed in the output files are easily understood. Model assessment result validity tests use *in vitro* and *in vivo* exercises to assert that a quality model's values do represent negative or positive additions to the software system. Table 6.6 presents the test cases discussed in this chapter, which category they fall into, and which questions and metrics they are associated with.

Table 6.6: Test Case Associations

Test Case	Category	Assoc. questions	Evaluated Metrics
TC-01	Construction/implementation effort	Q01-01	M01-M04, M10
TC-02	Construction/implementation effort	Q01-02	M05-M08
TC-03	Construction/implementation effort	Q01-03	M09, M11
TC-04	Construction/implementation effort	Q02-01	M12
TC-05	Construction/implementation effort	Q02-02	M17, M18
TC-06	Model understandability	Q01-04	M16
TC-07	Assessment result validity	Q01-05	M13-M15
TC-08	Assessment result validity	Q02-03	M19

6.3 Test Motivations

The tests preformed in this chapter are exercises which verify the functionality of PIQUE instances. While formal experimentation would produce much stronger data regarding the analysis of quality model process improvements, test cases are used because the quality models presented in chapter 2 function too differently from each other and PIQUE to be able to compare with acceptable internal and external validity in the context of the stated goals. For example, many models do not feature derivation, cannot be operationalized, or are no longer supported and cannot be experimented upon. Instead, to analyze a process of generating, validating, and operationalizing quality models for the purpose of improvement with respect to effort investment, experimentation, and collaborative opportunity, metric data comes from tests run on PIQUE quality model instances. A simple interpretation algorithm is then used to evaluate the success or failure of the goals for the given PIQUE instance.

This approach leads to a fault in external validity because the metrics are tied to a single PIQUE model design instance; however, given the rarity of platforms like PIQUE and the need for groundwork in its domain, this fault is considered acceptable; that is, the conclusions drawn from the test case results do not look to make claims of generalization, but rather look to show that improvement is achieved in the focused contexts of the goals stated at the beginning of this chapter, and that the platform has valuable offerings to further research in the domain of quality modeling.

6.4 Model Construction and Implementation Effort Tests

TC-01: Derive a C# Model Using Default Mechanisms

Motivation Test case 01 is an exercise in model derivation effort when using the default mechanisms provided by PIQUE. The test data is used to help evaluate the

components of goal G01 regarding the process of generating a quality model with respect to effort investment. The GQM question relevant to this test case is Q01-01, quantified using metrics M01-M04 and M10.

Given that many quality model approaches do not feature custom derivation, this test verifies that PIQUE supports derivation within a reasonable bound of effort. The test also provides a baseline for the additional effort needed to derive a model with modified mechanisms.

Test Design and Results The test is performed by installing PIQUE as a library resource, connecting third party C# static analysis tools, writing a C# quality model design file, preparing a a benchmark repository of 44 C# open source projects, and running the derivation process. The Roslynator tools, ISO/IEC 25010 model description, and benchmark repository from section 5.2 are used. Throughout the process, the effort taken to accomplish each task are recorded as metrics shown in table 6.7 with effort data rounded up to the nearest whole man-hour or man-hour-day.

Table 6.7: TC-01 metric results

Metric	Description	Value
M01	Man-hours taken to install PIQUE as a library resource	$<$ 1 hr.
M02	Man-hours taken to connect static analysis tools to PIQUE	$\sim$ 40 hr.
M03	Man-hours taken to design a quality model description *.json* file	$\sim$ 40 hr.
M04	Man-hours taken to prepare a benchmark repository	$\sim$ 16 hr.
M10	Time taken to run model derivation (benchmark repo size = 44)	1 hour

TC-02: Derive a C# Model Using Modified Mechanisms

Motivation Test case 02 is an exercise in model derivation effort when modifying the mechanisms provided by PIQUE. Given no other quality modeling platform presented in background chapters 2 and 3 allow modification of the model's internal mechanisms, the motivation of this test is to verify PIQUE does feature modification capabilities that work correctly and provide effort metrics. The resulting data is used to help evaluate the components of goal G01 regarding the process of generating a quality model with respect to effort investment and experimentation. The GQM question relevant to this test case is Q01-02, quantified using metrics M05-M08.

Test Design This test case contains a sub-test for each modifiable functional mechanism: normalization, utility function, weighting function, and the evaluation function. To verify a modified mechanism functions correctly, a control C# model is derived using no modified mechanisms. Each sub-test derives another C# model with components identical to the control model except for the modified mechanism. The two models are then compared: if their data is the same apart from the fields the modified mechanism should affect, the test passes and effort metrics are recorded.

The sub-tests and control model use the ISO/IEC 25010 model description from section 5.2, a benchmark repository containing the alphabetical first five projects from appendix D, and the comparison matrices of appendix E. The default mechanisms are shown in table 6.8.

TC-02A: Modified Normalizer

Design Sub-test 02A modifies the default normalizer provided by PIQUE as italicized in table 6.9. The underlying idea behind the test is to create a new normalizer class in a language-specific PIQUE project and link the new normalizer

Table 6.8: TC-02 default mechanisms

Mechanism	Affected Node(s)	Strategy
normalization	measure nodes	lines of code
utility function	measure nodes	3-threshold linear interpolation
weighting function	factor nodes	AHP
evaluation function	factor nodes	weighted sums
	measure nodes	count of findings

to one measure node while all other nodes use the default normalization strategy. If the resulting derived model displays different, non-normalized thresholds for only the modified measure node compared to the control model, a valid modification of the normalization mechanism has been achieved and effort metric M05 can be recorded.

Table 6.9: TC-02A normalization modification

Mechanism	Affected Node(s)	Strategy
normalization	all measure nodes *except one*	lines of code
normalization	*Format Smells measure node*	*do not normalize*
utility function	measure nodes	3-threshold linear interpolation
weighting function	factor nodes	AHP
evaluation function	factor nodes	weighted sums
	measure nodes	count of findings

To conduct this sub-test, a new class is made in PIQUE-CSHARP named

`NoNormalizer` which implements the `INormalizer` interface. By implementing the `INormalizer` class, the `normalize()` function must be defined in `NoNormalizer`. The strategy chosen here, as suggested by the class name, is to simply return the measure's in-value. This causes an effect of no normalization occurring only in nodes defined to use the "no normalizer" strategy.

To connect the strategy to a specific measure node, the quality model design *.json* file is modified. In the JSON measure object `Format Smells`, the following field is added.

```
Format Smells: {
    normalizer:
        pique.csharp.evaluation.NoNormalizer,
    ...
}
```

The sub-test passes if the new threshold values for the *Format Smells* measure node are the minimum, mean, and maximum number of diagnostic findings found in the benchmark repository (these values are not normalized by lines of code).

Results Table 6.10 shows the comparison of the threshold results for the measure *Format Smells* after model derivation. Specifically, the values represent the minimum, mean, and maximum number of static analysis tool findings of the diagnostics connected to *Format Smells* across the five benchmark projects. The threshold values for all other measures remained the same between the two models. The effort metric is recorded in table 6.11.

TC-02B: Modified Utility Function

Table 6.10: TC-02A *Format Smells* thresholds results

Control Model	No Normalizer Strategy	Sub-test Passed?
$t_1 = 0.00107901$	$t_1 = 36.0$	✓
$t_2 = 0.04586979$	$t_2 = 5176.0$	✓
$t_3 = 0.08338711$	$t_3 = 11398.0$	✓

Table 6.11: TC-02A metric result

Metric	Description	Value
M05	Man-hours taken to modify a default normalization function	1 hour

Design Sub-test 02B modifies the strategy used during the benchmarking phase that generates utility functions as shown in table 6.12 in italics. By default, PIQUE uses the benchmark repository to collect measure values across every benchmark project, removes outliers, and calculates three thresholds for each measure representing the min, median, and max values seen. Linear interpolation is then applied to the thresholds to output a benchmarked utility value of a given measure.

The modified slope utility function used in this test will find the median measure values and return 1.0 if a measure is above or equal to the median and 0.0 if it is below. If the *threshold.json* file output by PIQUE has only one threshold and if the measure evaluation values represent the new expected binary output, the sub-test is successful and metric M06 can be recorded.

To construct the test, a new class is made in PIQUE-CSHARP named `LinearUtility` implementing the `IUtilityFunction` interface. `LinearUtility` is then written to extract the median values of the measures of all nodes across all benchmark projects (after normalization but before utility functions are applied). `LinearUtility`

Table 6.12: TC-02B utility function modification

Mechanism	Affected Node(s)	Strategy
normalization	measure nodes	lines of code
utility function	measure nodes	*binary over-under median*
weighting function	factor nodes	AHP
evaluation function	factor nodes	weighted sums
	measure nodes	count of findings

also defines its `utilityStrategy()` method to return 1.0 if its input is above the measure's threshold, and 0.0 otherwise. To alert PIQUE of the new class, the global config field in the quality model design file is updated to configure the change in benchmarking strategy:

```
{
  name: CSharp ISO25k Quality Model,
  global_config: {
    utility_strategy: LinearUtility
  },
  ...
}
```

Finally, the quality model is used to assess the same system as assessed by the control model. The results–a few examples shown in table 6.13–match the prediction: all measure values are flattened to either zero or one depending if their assessed values were below or above the benchmarked mean. All other non-dependent quality model values stayed the same. The effort metric is recorded in table 6.14

Table 6.13: TC-02B utility function results

Measure	Control Model	Binary Utility	Sub-test Passed?
Format Smells	$t_1 = 0.0010$		✓
	$t_2 = 0.4586$	$t_1 = 0.4586$	✓
	$t_3 = 0.8338$		✓
	$value = 0.2740$	$value = 0.0$	✓
Funct. Syn. Smells	$t_1 = 9.6 * 10^{-4}$		✓
	$t_2 = 0.0013$	$t_1 = 0.0013$	✓
	$t_3 = 0.0014$		✓
	$value = 0.9520$	$value = 1.0$	✓

Table 6.14: TC-02B metric result

Metric	Description	Value
M06	Man-hours taken to modify a default utility function	1 hour

TC-02C: Factor Weighting Modification

Design Sub-test 02C modifies the weighting strategy used during model derivation. The default function uses the analytical hierarchy process with linguistic values as described in chapter 4.2. The modification strategy will put equal weighting across each node's children for every factor node in the hierarchy as represented by italics in table 6.15. A successful modification is easy to verify: if, for each factor, the derived weights of the factor equals $w_i = \frac{1}{|children|}$, the sub-test passes and metric M07 is recorded.

The test is designed by creating a new class in PIQUE-CSHARP named `EqualWeighter` which implements the `IWeighter` interface. `EqualWeighter` then

Table 6.15: TC-02C factor weighting modification

Mechanism	Affected Node(s)	Strategy
normalization	measure nodes	lines of code
utility function	measure nodes	3-threshold linear interpolation
weighting function	factor nodes	*all children equal*
evaluation function	factor nodes	weighted sums
	measure nodes	count of findings

overrides the `elicitateWeights()` interface method instructing PIQUE to set the weights of every factor to the average value of the number of the factor's incoming children.

Results The results, as shown in table 6.16, verify the expected difference in factor weighting between the two models. All other independent values stay the same. The effort metric is recorded in table 6.17

Table 6.16: TC-02C factor weighting results

Control Exp. Weights	EqualWeighter Weights	Sub-test Passed?
Compatibility = 0.0537	Compatibility = 0.1428	✓
Maintainability = 0.1822	Maintainability = 0.1428	✓
Perf. Efficiency = 0.151	Perf. Efficiency = 0.1428	✓
Portability = 0.0816	Portability = 0.1428	✓
Reliability = 0.4327	Reliability = 0.1428	✓
Security = 0.0156	Security = 0.1428	✓
Usability = 0.0831	Usability = 0.1428	✓

Table 6.17: TC-02C metric result

Metric	Description	Value
M07	Man-hours taken to modify a default weighting function	1 hour

TC-02D: Modified Evaluation Function

Design Sub-test 02D modifies the evaluation function used by the quality model during product assessment. By default, all factor nodes evaluate using the weighted sums of their child node values, and measure nodes evaluate using a count of its relevant findings (after applying normalization and utility functions).

This sub-test case simulates a potential use of QATCH in future research. Consider the topic of quality assessment in the domain of security. A critical software vulnerability has just been announced in a news letter that attackers will likely target on a practitioner's system. The vulnerability is detectable by static analysis tools. The practitioner states the potential effects of this flaw existing on their system is so dangerous that the quality assessment system needs to score any measure node aware of the vulnerability finding to 0.0 if the vulnerability is detected.

This sub-test simulates that Diagnostic node *RCS1206* is the static analysis diagnostic relevant to the critical vulnerability. As italicized in table 6.18, this sub-test changes the evaluation function of measure nodes with critical diagnostic *RCS1206* as a child to evaluate to 0 if one or more findings of *RCS1206* occurs during product assessment.

The sub-test is written by creating a new class in PIQUE-CSHARP named `SecurityEvaluator` which implements the `IEvaluator` interface. `SecurityEvaluator` is then linked to every measure node in the quality model design file. The pseudocode in the `SecurityEvaluator.evalStrategy()` method is as follows:

Table 6.18: TC-02D evaluation strategy modification

Mechanism	Affected Node(s)	Strategy
normalization	measure nodes	lines of code
utility function	measure nodes	3-threshold linear interpolation
weighting function	factor nodes	AHP
evaluation function	factor nodes	weighted sums
	measure nodes	*0 if critical diagnostic found*

```
if numFindings(RCS1206) > 0:
    set this.measure.evaluation to 0.0
else:
    run DefaultEvaluator.evalStrategy()
```

Product assessment is performed on a system named FASTER[1]–an open source C# system with 62,000 lines of code–which is known to have at least one finding of *RCS1206*. The *RCS1206* diagnostic is connected strictly to the *Functional Syntax Smells* measure; thus, the sub-test passes if the *Functional Syntax Smells* measure's node value has a non-zero value when using the default evaluation strategy and has a value of 0.0 using the modified evaluation strategy. All other independent node values must stay the same.

Results Table 6.19 shows the difference in evaluation when one model is given instruction to use the unique `SecurityEvaluator` evaluation strategy. A previously high scoring measure now evaluates to 0.0 strictly because of a single finding while the

[1] `https://github.com/microsoft/FASTER`

other independent model values had matching values. The effort metric is recorded in table 6.20.

Table 6.19: TC-02D measure node result

Measure	Default Evaluation	Modified Evaluation	Sub-test Passed?
Fun. Syntax	"value": 0.9520	"value": 0.0	✓

Table 6.20: TC-02D metric result

Metric	Description	Value
M08	Man-hours taken to modify a default evaluation function	1 hour

TC-03: Operationalize a C# Model

Motivation Test case 03 is a validation exercise targeting quality model operationalization effort. It is necessary to verify that PIQUE-derived quality models can be operationalized, and data regarding effort to operationalize is needed. This test case addresses GQM question Q01-03 by producing metrics M09 and M11.

Test Design and Results To verify operationalization success, this test runs PIQUE quality assessment on a C# project and verifies the assessment output scores are correct given known, detectable findings in the product under assessment. The ISO/IEC 25010 C# quality model derived from section 5.2 is used as the assessment model, and a custom C# solution is set as the product under assessment. The product under assessment has 58 total lines of code, contains one *.cs* class, and is constructed to contain exactly one finding detectable by the quality model as shown in the following code snippet.

```
{
    ...
    var rnd = new Random();
    byte[] buffer = new byte[16];
    rnd.NextBytes(buffer);
    return BitConverter.ToString(buffer);
    ...
}
```

This block of code triggers the *SCS0005* diagnostic from the Security Code Scan tool.[2] In the quality model, diagnostic *SCS0005* is a child of only the measure *Encryption Smells*, and *Encryption Smells* is a child of only the product factor *Security*. Given *SCS0005* should be the only detected finding after assessment and all findings has negative impact on the quality score, the test is designed to pass if the following assertions are met:

- The *Encryption* measure's quality score is 0.0.
- The *Encryption* measure contains exactly one finding named *SCS0005*.
- The *Security* product factor's quality score is less than 1.0.
- All other product factor's have a quality score of 1.0.

If the test passes, effort metrics M09 and M11 are recorded.

Assessment is run using the packaged form of PIQUE-CSHARP[3] (chapter 5) with the quality model location and system under assessment directory as arguments. The

[2] `https://security-code-scan.github.io`
[3] PIQUE-CSHARP packages and deploys as a runnable *.jar* file.

assessment output file is used to run test assertions as shown in table 6.21 with effort metrics provided in table 6.22.

Table 6.21: TC-03 assessment results

Model Node	Output JSON Data	Test Passed?
Encryption Measure	positive_impact: false	✓
	parents: [Encryption]	✓
	findings: [SCS0005 @line 13]	✓
	evaluation: 0.0	✓
Security Product Factor	evaluation: 0.0	✓
All Other Product Factors	evaluation: 1.0	✓

Table 6.22: TC-03 metric results

Metric	Description	Value
M09:	Man-hours taken to operationalize a derived quality model.	1 hour
M11:	Time taken to run system quality assessment.	$<$ 1 hour

TC-04: Practitioner Interaction Effort

Motivation Thus far, tests and validations have been performed from the perspective of the researcher, but in order to address goal G02, effort exercises need to be performed from the practitioner's perspective. Test case 04 utilizes a joint effort with a local company to produce data regarding effort required to express subjective quality opinion usable by PIQUE model derivation operations. The gathered data helps answer question Q02-01 through use of metric M12.

Test Design and Results The test involves interactions with a local company in which the basic PIQUE quality model concepts are explained, then a practitioner fills in an AHP comparison matrix.[4] The quality model used is the ISO/IEC 25010 C# quality model design from chapter 5, and the AHP matrix used represents the importance ratings from the quality aspect to total quality index layer.

The resulting comparison matrix as entered by the practitioner is given in table 6.23 where the cell values are linguistic comparators of the importance of the row entry compared to the column entry. The acronyms represent {VL: very low. SL: somewhat low. EQ: equal. SH: somewhat high. VH: very high}. The effort metric M12, recorded after a successful table entry by the practitioner, is shown in table 6.24. Note that expended effort for other AHP matrices can vary greatly depending on the context and environment. This threat to validity is addressed further in the discussions of chapter 7.

Table 6.23: TC-04 comparison matrix result

TQI	Comp.	Maint.	Perf.	Port.	Rel.	Sec.	Use.
Comp.	-	SL	SL	SL	VL	VH	EQ
Maint.	-	-	EQ	SH	SL	VH	SH
Perf.	-	-	-	SH	SL	VH	EQ
Port.	-	-	-	-	VL	VH	EQ
Rel.	-	-	-	-	-	VH	SH
Sec.	-	-	-	-	-	-	VL
Use.	-	-	-	-	-	-	-

[4] AHP matrices are discussed in section 4.2.

Table 6.24: TC-04 metric result

Metric	Description	Value
M12	Time by one practitioner to enter subjective quality definitions	2 hours

TC-05: External System Integration

Motivation An aspect of goal G02 is the process of quality model operationalization from the point of view of software development practitioners with the associated question Q02-02, *How difficult is it to integrate quality model assessment into an external, continuous integration system?* This is a valuable question to address due to the importance of a quality assessment approach being compatible and usable by other users. The metrics chosen to quantify Q02-02 are M17 and M18.

Metric M17, *Number of external dependencies needed for actualization*, is used as a way to represent the compatibility of the assessment engine when considering different operating systems and unknown environments. The GQM interpretation algorithm used in section 7.2 states that goal G02 has failed if the operationalization process requires any more than one specific external dependency: the Java Runtime Environment.

The boolean metric M18, *Failure of assessment does not interfere with other system processes*, is chosen to address a crucial aspect of usability and acceptability of quality assessment engines expressed by practitioners: quality assessment should not interfere with normal software development processes already in place. Given the persisting distrust of software quality processes in practice [43], it is vital these processes do not cause problems with the system under assessment. Quality assessment–a vital component of software engineering processes–will likely not find

acceptance in practice if there is concern its adoption will cause additional costs and concerns.

Description This test runs an exercise deploying a language-specific operationalization of PIQUE into the environment of a local company's software development life cycle. After successful deployment and product quality assessment, the number of external dependencies required and obstruction of the normal development life cycle are evaluated.

To deploy PIQUE-CSHARP, the system is packaged as a Java *.jar* file. The JAR's manifest defines a main class, `Assessment` which takes a *.properties* file as input and uses the PIQUE Runner component to activate quality assessment on a target system. The *.properties* file defines all necessary configurations such as the location of the system under evaluation, the quality model file to use, and where to place assessment result files. All necessary external static analysis tools are packaged with the *.jar* file as a resource.

The industrial partner interacting with this exercise uses GitLab[5] for their development. To integrate PIQUE-CSHARP quality assessment, a simple, one-line, PowerShell script build step within their continuous integration (CI) pipeline process is added as follows:

```
java -jar piqueCSharpAssessment.jar config.properties
```

Two cases are checked in order for the test to pass:

- Quality assessment is successfully run as part of the CI pipeline and a quality assessment output file is generated with expected values.

[5] `https://gitlab.com/`

- Quality assessment fails, but the other CI pipeline processes complete without errors.

Assessment is run twice: PIQUE-CSHARP normal assessment, and PIQUE-CSHARP assessment with a bug intentionally introduced causing the assessment process to fail with errors. If the tests pass, metric values are recorded in table 6.25.

Results Given that packaged Java applications are designed to run on any operating system, the only necessary dependency needed was Java (version 8+). Because assessment is run as as single pipeline step whose results no other process depends on, the CI pipeline was configured to continue its standard processes regardless of the success or failure of the quality assessment step.

Table 6.25: TC-05 metric results

Metric	Description	Value
M17	Number of external dependencies needed for actualization	1
M18	[True | False] Failure of assessment does not interfere with other system processes	True

6.5 Model Understandability Tests

TC-06: Investigate Model Output Accessability

Motivation Test case 06 addresses G01's goal of improvement of quality model processes with respect to effort investment as stated by question Q01-04: *Do the models produced facilitate ease of evaluation by researchers?* The output file generated by a quality model's processes is the most accessible artifact used by researchers for process evaluation. By comparing the amount of information a researcher may want

to see to what is actually exposed, metric data can be obtained to quantify effort investment concepts.

Test Design and Results To evaluate the accessability of model process information, a metric is introduced, exposure, defined to be the ratio of the total number of model-related information used during runtime compared to the amount of information exposed by the output file. Specifically, this information is the internal mechanisms, objects, and primitive values used by the quality engine during model derivation or product quality assessment.

The test case is run by deriving a model and running quality assessment on a project while using the debugger to record the mechanisms, objects, and evaluations relevant to quality processes at each step of operation. These values are compared to the information presented by the derivation and assessment output files. The data is shown in table 6.26 and discussion of the exposure ratio interpretation is given in chapter 7.

The model is derived using the ISO/IEC C# quality model description used in section 5.2, the Roslynator tools described in appendix C, the benchmark repository of appendix D, and the comparison matrices of appendix E. The product under assessment is a custom C# solution with one class, 58 lines of code, and an injected finding detectable by Security Code Scan with diagnostic name *SCS0005*.

Table 6.26: PIQUE output exposure test

Internal Mechanisms, Objects, Values	Exposed?
Static analysis tool objects	
Static analysis tool names	✓
Benchmarking strategy name	✓
Benchmarking strategy logic	

Benchmarking output data	✓
Weighting strategy name	✓
Weighting strategy logic	
Weighting output data	✓
Quality model	✓
Model node name	✓
Model node final value	✓
Model node evaluation strategy name	✓
Model node evaluation strategy logic	
TQI node	✓
TQI incoming weights	✓
Quality Aspect nodes	✓
Quality Aspect incoming weights	✓
Product Factor nodes	✓
Product Factor incoming weights	✓
Measure nodes	✓
Measure node +/− impact	✓
Measure node normalizer	✓
Measure node incoming Diagnostics	✓
Measure node utility function thresholds	✓
Measure node value before normalization	
Measure node value before utility function	
Diagnostic nodes	✓
Diagnostic associated tool name	✓
Diagnostic incoming Findings	✓
Finding nodes	✓

Finding file path	✓
Finding line number	✓
Finding character number	✓
Finding numerical value	✓
Finding additional info (e.g. severity)	✓
Normalizer object	
Benchmarker object	
Evaluator object	
Project	✓
Project name	✓
Project lines of code	✓
Project path	✓
Project's quality model	✓

The resulting ratio is $M16 = exposure = \frac{exposed}{total} = \frac{35}{43} = 0.81$

6.6 Model Assessment Result Validity Tests

TC-07: Introduce In Vitro Product Changes

Motivation and Design In order to analyze the process of validating and operationalizing quality models, a platform has been presented that can facilitate comparisons of quality modeling approaches, but the platform must be tested for accuracy in quality assessment. Question Q01-05, *are the models produced valid*, is addressed using three *in vitro* exercises: (1) validate that flaws introduced to a system cause a negative change in quality; (2) validate that fixes introduced to a system cause a positive change in quality; and (3) validate that a subjective change in quality priorities are accurately reflected by the system. These three exercises are

separated into three sub-tests: TC-07A, TC-07B, and TC-07C which generate data values for metrics M13, M14, and M15 respectively.

Control Values The control assessment provides an environment and assessment values as a base line of comparison for the results of TC-07A through TC-07C. The model for assessment is derived using the ISO/IEC C# quality model description used in section 5.2, the Roslynator tools described in appendix C, the benchmark repository of appendix D, and the comparison matrices of appendix E. All default mechanisms are used: 3-threshold linear interpolation utility function, AHP for subjective factor weighting, and normalization by lines of code.

The C# project under assessment is hand-tailored for sake of *in vitro* testing. One unique diagnostic finding is injected for the measures *Format Smells*, and *Resource Handling Smells*. Filler lines of code are added or removed to keep the lines of code the same across each sub-test. In total, the hand-tailored C# solution has 114 lines of code, one class, and a method for each injected finding.

The model node values resulting from this assessment are shown in table 6.27.[6] Note that the quality scores of the measures with findings will be low for this project due to flaws injected in to a project with very few lines of code: when normalized by lines of code, even one finding will likely bring a measure value below the minimum benchmarked threshold.

Table 6.27: TC-07 control assessment values

Node Type	*Node Name*	*Value*
TQI	Total Quality	0.8691
Quality Aspect	Perf. efficiency	0.7771
-	Portability	0.7808

[6]For brevity, incoming edge weight values are not included, but can be found in appendix F.1.

Table 6.27: TC-07 control assessment values

Node Type	*Node Name*	*Value*
-	Maintainability	0.9392
-	Compatibility	0.8411
-	Reliability	0.8778
-	Security	0.8856
-	Usability	0.9399
Product Factor	Func. Syntax	1.0
-	Format	0.8196
-	Resource Handling	0.0895
-	DT Integrity	1.0
-	Documentation	1.0
-	IO Handling	1.0
-	Exception Handling	1.0
-	Encryption	1.0
-	Structure	1.0
Measure	EH Smells	1.0
-	FS Smells	1.0
-	Encryption Smells	1.0
-	Doc. Smells	1.0
-	Format Smells	0.8196
-	DTI Smells	1.0
-	RH Smells	0.0895
-	IOH Smells	1.0
-	Structure Smells	1.0

TC-07A: Inject Flaws Sub-test TC-07A evaluates whether a model generated by PIQUE correctly responds to the introduction of poor quality lines of code. Using the environment of the control assessment, one additional code smell is injected in the C# project associated to the following measures: *Exception Handling Smells*, *Encryption Smells*, *Documentation Smells*, *DTI Smells*, and *Structure Smells*. The test records the change in node values given the introduction of flaws as shown in table 6.28.

Given the quality aspect to product factor layer is fully connected, the test expects all quality aspects values to decrease. Each product factor has exactly one child–a measure sharing the same name–so the test expects only the product factors and measures associated with *Exception Handling*, *Encryption*, *Documentation*, *DTI*, and *Structure* to decrease in value. All other values should stay the same. The resulting metric data from this test is given in table 6.29.

Table 6.28: TC-07A node values after introducing flaws

Node Type	*Node Name*	*Control Value*	*Value After*
TQI	Total Quality	0.8691	**0.2742**
Quality Aspect	Perf. efficiency	0.7771	**0.3082**
-	Portability	0.7808	**0.3739**
-	Maintainability	0.9392	**0.3627**
-	Compatibility	0.8411	**0.2834**
-	Reliability	0.8778	**0.1795**
-	Security	0.8856	**0.3247**
-	Usability	0.9399	**0.3989**
Product Factor	Func. Syntax	1.0	1.0
-	Format	0.8196	0.8196
-	Resource Handling	0.0895	0.0895

Table 6.28: TC-07A node values after introducing flaws

Node Type	*Node Name*	*Control Value*	*Value After*
-	DT Integrity	1.0	**0.0**
-	Documentation	1.0	**0.0**
-	IO Handling	1.0	1.0
-	Exception Handling	1.0	**0.0**
-	Encryption	1.0	**0.0**
-	Structure	1.0	**0.0**
Measure	EH Smells	1.0	**0.0**
-	FS Smells	1.0	1.0
-	Encryption Smells	1.0	**0.0**
-	Doc. Smells	1.0	**0.0**
-	Format Smells	0.8196	0.8196
-	DTI Smells	1.0	**0.0**
-	RH Smells	0.0895	0.0895
-	IOH Smells	1.0	1.0
-	Structure Smells	1.0	**0.0**

Table 6.29: TC-07A metric result

Metric	Description	Value
M13:	The change in quality assessment score decreases after introducing flaws.	True

TC-07B: Inject Fixes Sub-test TC-07B evaluates whether a model generated by PIQUE correctly responds to the introduction of quality fixes in a software system.

Given the environment of the control assessment, the code smell relevant to the *Resource Handling* measure is fixed in the C# project. The test records the change in node values given these quality improvements as shown in table 6.30.

Given the quality aspect to product factor layer is fully connected, the test expects all quality aspects values to increase. Each product factor has exactly one child–a measure sharing the same name–so the test expects only the product factor and measure associated with *Resource Handling* value to increase. All other values should stay the same. Table 6.31 shows the resulting metric data.

Table 6.30: TC-07B node values after fixing a flaw

Node Type	*Node Name*	*Control Value*	*Value After*
TQI	Total Quality	0.8691	**0.9893**
Quality Aspect	Perf. efficiency	0.7771	**0.9963**
-	Portability	0.7808	**0.9948**
-	Maintainability	0.9392	**0.9684**
-	Compatibility	0.8411	**0.9960**
-	Reliability	0.8778	**0.9978**
-	Security	0.8856	**0.9966**
-	Usability	0.9399	**0.9683**
Product Factor	Func. Syntax	1.0	1.0
-	Format	0.8196	0.8196
-	Resource Handling	0.0895	**1.0**
-	DT Integrity	1.0	1.0
-	Documentation	1.0	1.0
-	IO Handling	1.0	1.0
-	Exception Handling	1.0	1.0

Table 6.30: TC-07B node values after fixing a flaw

Node Type	*Node Name*	*Control Value*	*Value After*
-	Encryption	1.0	1.0
-	Structure	1.0	1.0
Measure	EH Smells	1.0	1.0
-	FS Smells	1.0	1.0
-	Encryption Smells	1.0	1.0
-	Doc. Smells	1.0	1.0
-	Format Smells	0.8196	0.8196
-	DTI Smells	1.0	1.0
-	RH Smells	0.0895	**1.0**
-	IOH Smells	1.0	1.0
-	Structure Smells	1.0	1.0

Table 6.31: TC-07B metric result

Metric	Description	Value
M14:	The change in quality assessment score increases after introducing improvements.	True

TC-07C: Modify Subjectivity Given that subjective quality awareness is a vital part of the quality model derivation process, the validity of its mechanism used in PIQUE should be evaluated. By default, PIQUE uses the analytical hierarchy process in the form of handwritten comparison matrices to introduce subjectivity to the model's factor edges in an automated fashion.

Sub-test TC-07C generates data regarding the difference in quality model output scores when changing the comparison matrix configurations at the quality aspect level. The control comparison matrix shown in table 6.32 values security above all other quality factors while the rest are set as neutral priority with each other. To test the opposite preference, the matrix shown in table 6.33 modifies security as least meaningful in the context of total software quality while the other quality aspects are valued equally with each other.[7]

Table 6.32: TC-07C TQI-quality aspect comparison matrix (control)

TQI	Comp.	Maint.	Perf.	Port.	Rel.	Sec.	Use.	Derived Weights
Comp.	-	EQ	EQ	EQ	EQ	**VL**	EQ	0.0667
Maint.	-	-	EQ	EQ	EQ	**VL**	EQ	0.0667
Perf.	-	-	-	EQ	EQ	**VL**	EQ	0.0667
Port.	-	-	-	-	EQ	**VL**	EQ	0.0667
Rel.	-	-	-	-	-	**VL**	EQ	0.0667
Sec.	-	-	-	-	-	-	**VH**	**0.6**
Use.	-	-	-	-	-	-	-	0.0667

Next, a system under assessment is designed to have strictly one unique *Encryption Smells* finding. An encryption finding is used because the measure, *Encryption Smells* measures the *Encryption* product factor which has the highest weighted impact on the *Security* quality aspect according to the quality model used for this sub-test.

Given the quality aspect weights of tables 6.32 and 6.33, the test expects the

[7]The derived weights come from running the model derivation process on a small benchmark repository using the alphabetical first five systems from appendix D, but otherwise the same environment as the TC-07 control model is used.

Table 6.33: TC-07C TQI-quality aspect comparison matrix (modified environment)

TQI	Comp.	Maint.	Perf.	Port.	Rel.	Sec.	Use.	Derived Weights
Comp.	-	EQ	EQ	EQ	EQ	**VH**	EQ	0.1636
Maint.	-	-	EQ	EQ	EQ	**VH**	EQ	0.1636
Perf.	-	-	-	EQ	EQ	**VH**	EQ	0.1636
Port.	-	-	-	-	EQ	**VH**	EQ	0.1636
Rel.	-	-	-	-	-	**VH**	EQ	0.1636
Sec.	-	-	-	-	-	-	**VL**	**0.0182**
Use.	-	-	-	-	-	-	-	0.1636

Table 6.34: TC-07C quality evaluations given different AHP security preferences

Node Name	*Control Value*	*Value After*
Total Quality Index	**0.8231**	**0.9731**
Perf. efficiency	0.9734	0.9734
Portability	0.9739	0.9739
Maintainability	0.9875	0.9875
Compatibility	0.9756	0.9756
Reliability	0.9884	0.9884
Security	0.7196	0.7196
Usability	0.9689	0.9689

injected finding to cause the TQI score of the modified test to be much higher than the TQI score of the control test. The values of the quality aspects (performance efficiency, portability, maintainability, etc.) are expected to not change because the difference in derived weights only applies to the weighted sum evaluation of the Total

Quality Index node value. The evaluated factor node values are given in table 6.34 with table 6.36 showing the metric data recorded.

TC-08: Quality Output Trustability

Motivation As expressed by goal G02 and question Q02-03, the final, industry-focused test case relates to the trustability of the quality values output after assessment on an industry partner's software system. The perspective taken in this thesis is that quality values alone are meaningless. A quality value only has meaning when given context. Specifically, this context is the change in quality assessment values relative to their past values. Given the output values from quality model assessments still remain distrusted by practitioners [43], an exercise showing a PIQUE-built model has capability to provide meaningful, trustable, or relatable values is an encouraging step forward.

Test Design and Results Due to the subjective nature of trust, this exercise uses interaction with a local, industry partner. The test compares PIQUE quality assessment results against practitioner opinion regarding the change in quality scores over a series of product changes.

The model for assessment is derived using the ISO/IEC C# quality model description used in section 5.2, the Roslynator tools described in appendix C, the benchmark repository of appendix D, and the comparison matrices of appendix E.[8] All default mechanisms are used: 3-threshold linear interpolation utility function, AHP for subjective factor weighting, and normalization by lines of code. After deriving the model, the PIQUE assessment engine is packaged as a *.jar* file and introduced to the industry partner's source code management system.

[8]The comparison matrix at the TQI to quality aspect level represents the subjective quality opinions of the industry partner of this test.

Quality assessment is run over a collection of historical product changes. These histories, shown in table 6.35, are selected by the practitioner with the intention that each history, subjectively, represents notable positive changes in quality. Detailed date and commit information is not provided to maintain anonymity. The right columns of table 6.35 show the PIQUE score of the start commit, the finish commit, and whether PIQUE also output a positive change in quality score.

Table 6.35: Commit histories of subjectively perceived improvement

Start	*Finish*	*# Commits*	*S-Quality*	*F-Quality*	*Pos. slope?*
Oct. 22	Dec. 04	157	0.4335	0.4102	
Jan. 21	Feb. 10	171	0.4087	0.4186	✓
Jan. 30	Apr. 15	608	0.4082	0.3840	
Mar. 25	Mar. 27	26	0.3826	0.3836	✓

Table 6.36: TC-08 metric result

Metric	Description	Value
M19:	[True | False] The slope of assessed values over time matches practitioner opinion.	False

The relevant metric to this test, M19, evaluates to true if all commit history exercises have the same slope as the practitioner's opinion. Given the histories of October 22-December 04 and January 30-April 15 did not return a desired positive slope of quality change, M19 evaluates to *false* in the context of this test case. Practitioner opinion of quality change versus objective quality change may not always align, so the results of this test case deserve further discussion, given in section 7.2.

6.7 Summary of Results

The test cases of this chapter utilize PIQUE to generate data for use in evaluating the GQM components described in chapter 1.2. The metrics are summarized in the following table and used to drive the discussion of chapter 7. Chapter 7 uses improvement algorithms parameterized by the metrics of this chapter to assess the success or failure of the given GQM goals.

Table 6.37: Effort Metrics

Metric	*Description*	*Value*
M01	Man-hours taken to install PIQUE as a library resource.	$<$ 1 hour
M02	Man-hours taken to connect static analysis tools to PIQUE.	1 week
M03	Man-hours taken to design a quality model description *.json* file	1 week
M04	Man-hours taken to prepare a benchmark repository (n = 44).	2 days
M05	Man-hours taken to modify a default normalization function.	1 hour
M06	Man-hours taken to modify a default utility function.	1 hour
M07	Man-hours taken to modify a default weighting function.	1 hour
M08	Man-hours taken to modify a default evaluation function.	1 hour
M09	Man-hours taken to operationalize a derived quality model.	1 hour
M10	Time taken to run model derivation (benchmark repo size = 44).	1 hour
M11	Time taken to run system quality assessment.	$<$ 1 hour

Table 6.37: Effort Metrics

Metric	*Description*	*Value*
M12	Time taken by a practitioner to enter subjective quality definitions.	2 hours

Table 6.38: Quality Assessment Metrics

Metric	*Description*	*Value*
M13	The change in quality assessment score decreases after introducing flaws.	True
M14	The change in quality assessment score increases after introducing improvements.	True
M15	The change in quality assessment score after modifying subjective quality opinion reflects the new prioritization.	True

Table 6.39: Design Metrics

Metric	*Description*	*Value*
M16	Model output exposure.	0.81
M17	Number of external dependencies needed for actualization.	1
M18	[True | False] Failure of assessment does not interfere with other system processes	True

Table 6.40: Practitioner Subjectivity Metric

Metric	*Description*	*Value*
M19	[True | False] The slope of assessed values over time matches practitioner opinion.	False

Table 6.41: Requirement Metrics

Metric	*Description*	*Value*
M20	[True | False] The platform is open source.	True

CHAPTER SEVEN

DISCUSSION

Two closely related goals are presented in the opening chapter related to analyzing the process of generating and using quality models from the perspective of researchers and industry. To evaluate these processes, a platform for quality model generation and operationalization is presented in chapter 4 and actualized for the C# language in chapter 5. These structures are used to run a variety of exercises in chapter 6 to validate the system and generate metric data. The produced metrics are now reviewed in the context of the goals and questions originally asked along with a general discussion of the systems presented in this thesis.

This chapter uses the GQM approach to drive discussion due to the lack of comparability of the models presented in this thesis (chapters 2, 3, and 4) when considering effort investment, cost investment, experimentation capabilities, and collaborative opportunity. Given that operationalized quality modeling approaches are software products, the GQM approach provides a way to make comparisons using metrics and interpretation algorithms as described by Basili [6].

The interpretation algorithms used to evaluate improvement in this section are rudimentary and apply only to the model instance that generated the metrics; however, achieving goals of improvement for quality modeling processes–even with weak metrics–is valuable given the young state of research in the domain.

7.1 Goal 01: The Research Perspective

Using the goal/question/metric (GQM) paradigm of [6], goal one states,

"Analyze a process of generating, validating, and operationalizing quality

> models for the purpose of improvement with respect to effort investment, experimentation, and collaborative opportunity from the point of view of quality model researchers in the context of static software system analysis."

The following sections answer the questions relevant to goal G01 using simple interpretation algorithms. Finally, the results obtained for G01 are discussed using the evaluations of questions Q01-01 through Q01-04.

Model Generation Effort

Question Q01-01, *how much effort does it take to generate a model using default mechanisms*, is asked to analyze the process of model generation with respect to effort investment. Given this question is asked from the point of view of quality model researchers, the question is targeting the scenario of researchers desiring to generate a novel model that does not have satisfying supporting work in place to assist its construction. The relevant metric results from the test cases are M01-M04 and M10 as shown in table 7.1.

Q01-01 Interpretation Algorithm Improvement regarding model generation is interpreted in context of the Quamoco project due to their similarities in underlying mechanisms and model designs. The Quamoco project [42] involved a team size of over twenty researchers using an iterative development process that took over a year to complete an acceptable model. Given Quamoco is a recent, novel quality modeling attempt that tries to solve many of the same problems PIQUE considers, it can function as a base line of effort investment from which to improve upon.

Directly comparing the effort involved in novel model creation using PIQUE versus the experiences of the QUAMOCO team is a threat to internal and external

Table 7.1: Q01-01 metrics

Metric	Description	Value
M01	Man-hours taken to install PIQUE as a library resource.	$<$ 1 hour
M02	Man-hours taken to connect static analysis tools to PIQUE.	1 week
M03	Man-hours taken to design a quality model description *.json* file.	1 week
M04	Man-hours taken to prepare a benchmark repository.	2 days
M10	Time taken to run model derivation (benchmark repo size = 44).	1 hour

validity because the processes and rigor used in each approach vary greatly, so the interpretation algorithm provides a weak assessment of improvement, but it is still valuable considering the quality improvement paradigm of [6]. The improvement algorithm of listing 7.1 compares the sum of effort metrics M01-M04 and M10 to an effort window of 12 months. Twelve months or less of effort is chosen due to the effort of over one year realized by the Quamoco project.

Listing 7.1: Q01-01 interpretation algorithm

```
1 target = 12 months
2 repeat until best possible target achieved
3     if (M01 + M02 + M03 + M04 + M10)
4         < target then
5             method better than history
6             target = M01 + M02 + M03 + M04 + M10
7         >= target then
8             M_x = max(M01, M02, M03, M04, M10)
```

```
9                reduce effort necessary for M_x
```

Q01-01 Discussion From the metrics generated by test case TC-01 using PIQUE, $M01 + M02 + M03 + M04 + M10 < 3\,weeks$ resulting in a method better than the target history. Given the test case involved operationalizing a new model in C# with new tools and approaches, the tasks of identifying and connecting C# static analysis to a hand-made model description was the most time consuming task. The strength of utilizing PIQUE for new quality model derivations is its capability to automate the other processes necessary to operationalize a quality model. By only needing to define the language-specific or novel components involved in model derivation such as the C# tools and the model structure through either text-based entries or simple interface implementations, a notable effort-saving advantage is seen through use of PIQUE.

Model Generation Effort (Modified Mechanisms)

Question Q01-02, *how much effort does it take to generate a model using modified mechanisms*, is also asked to analyze the process of model generation with respect to effort investment; however, going beyond the scenario of question Q01-01, this question targets the point of view of quality model experimentation. For example, consider the scenario of an existing QATCH quality model, but a researcher desires to re-derive the model after modifying some of the internal mechanisms for experimental pursuits. The relevant metrics are ones that quantify the effort needed to modify the internal mechanisms: metrics M05 through M08 shown in table 7.2.

Q01-02 Interpretation Algorithm Improvement regarding modification of internal mechanisms is interpreted in the context of the features presented by the

Table 7.2: Q01-02 metrics

Metric	Description	Value
M05	Man-hours taken to modify a default normalization function.	1 hour
M06	Man-hours taken to modify a default utility function.	1 hour
M07	Man-hours taken to modify a default weighting function.	1 hour
M08	Man-hours taken to modify a default evaluation function.	1 hour

other quality models and frameworks presented in chapters 2 and 3. None of the reviewed approaches directly support mechanism modification. For the open source projects, one could modify the source code to alter an intended mechanism such as replacing linear threshold calculations with a more complex curve. However, given the reviewed approaches are designed to function as-is, such changes will often lead to a time consuming implementation introducing bugs and compilation errors.

Thus, the algorithm to express improvement is rudimentary: if any internal mechanism features modifiability in a modular manner non-destructive to the original mechanism implementation achievable in a finite amount of time, improvement is achieved. The pseudocode for this algorithm is given in listing 7.2.

Listing 7.2: Q01-02 interpretation algorithm

```
1 target = infinity
2 repeat until best possible target achieved
3     if (M05 || M06 || M07 || M08)
4         < target then
```

```
5               method better than history
6               target = min(M05, M06, M07, M08)
7           > target then
8               M_x = max(M05, M06, M07, M08)
9               reduce effort necessary for M_x
```

Q01-02 Discussion From the metrics generated by TC-02 using PIQUE, $M05 \vee M06 \vee M07 \vee M08 \leq 1\ hour \ll target$ resulting in a method better than the history. The comparison of features in historical models studied to the features of PIQUE is a notable threat to construct validity because the reviewed models were not created with the intention of acting as a platform for model design experimentation; rather, they present a single, concrete, operationalized model whose internal mechanisms are not intended to be modified.

Given this disparity of comparing apples to oranges, the improvement algorithm could also be presented as "*Is internal mechanism modification now featured and does it work correctly?*" Since this question targets the existence of a feature that previously did not exist, the mere presence of the feature is an improvement. Regardless of the algorithm's phrasing, improvement is realized.

Metrics M05 through M08 refer to distinctly four internal mechanisms: normalization functions, utility functions, weighting functions, and evaluation functions. These mechanisms were chosen after identifying the extent of mechanisms used in historical quality modeling efforts. However, different, unique, unknown mechanisms will reveal themselves as research continues in the domain of quality modeling that cannot currently be represented as metrics. The need to plan for unknown potential has been addressed through the modular design of PIQUE as discussed in section 4.1. While contribution to the source code of PIQUE would be required, development to

extend the system in this context is straightforward, requiring simple class additions to the CALIBRATION or EVALUATION components.

Model Operationalization Effort

Question Q01-03, *how much effort does it take to operationalize a model*, is asked to analyze the process of model operationalization with respect to effort investment. The question is asked from the point of view of anyone attempting to operationalize a quality model, whether for academic or industrial pursuits. A standard use case which attempts to answer this question is the integration of a packaged quality model into an external code base for the sake of outputting quality assessment values. The relevant metric from the test cases is M09, shown in table 7.3.

Table 7.3: Q01-03 metric

Metric	Description	Value
M09	Man-hours taken to operationalize a derived quality model	1 hour

Q01-03 Interpretation Algorithm Improvement regarding model operationalization is interpreted in the context of the modern quality models reviewed in sections 2.5 and 3 that are still supported. Modern, supported models are used for comparison in order to compare the operationalization of PIQUE models to other modeling attempts that are also still able to be operationalized on modern computers and languages.

Of these modern, supported models, Quamoco and SQALE[1] are notable successful examples in terms of operationalization. Although Quamoco has not seen updates in over five years to its GitHub repository, its Java-based intuitive GUI and

[1] as supported by SonarQube

operationalization wizard facilitates successful quality assessment using an 'out-of-the-box' Java quality model in under an hour. SQALE, through its integration with SonarQube, presents a streamlined, web-based experience that simply requires a user to point at the root directory of their project causing all other configurations to work behind the scenes if using their out-of-the-box quality model. Basic quality assessment through SonarQube's SQALE quality model can be achieved in mere minutes.

The interpretation algorithm of listing 7.3 considers of the experiences provided by Quamoco and SQALE in the context of improvement. Both frameworks offered better graphical usability, but improvement in this context will only be considered in terms of time-based effort. Given that SQALE presents quality assessment achievable in under 10 minutes, this value will be used as the initial target for improvement.

Listing 7.3: Q01-03 interpretation algorithm

```
1 target = 10 minutes
2 repeat until best possible target achieved
3     if M09
4         < target then
5             method better than history
6             target = M09
7         >= target then
8             reduce effort necessary for M09
```

Q01-03 Discussion The value of M09 shown in table 7.3 and the time realized to operationalize Quamoco and SQALE represent a threat to external validity because their values come from the experiences of this thesis' author's attempts. Given this author's extreme familiarity with operationalizing PIQUE models and general familiarity with running quality assessment using quality models similar to SQALE

and Quamoco, there is inherent bias behind the metric values so generalization of results is not possible. This threat is somewhat mitigated due to the fact that $M09 > target$, thus the PIQUE method did not show a value better than the history. The bias of PIQUE familiarity could cause the value of M09 to be lower than the general case, but since the metric was still not better than the history, the result can be interpreted as a sign that improvement regarding model operationalization was not achieved even with the bias of familiarity.

Question Q01-03 targets the nature of usability and accessability of a quality model's assessment. Historically, other quality model frameworks have put notable effort on user experience, so failing to achieve improvement in the context of Q01-03 was expected. Despite this lack of improvement, PIQUE still offers important steps forward in the domain of quality modeling. SonarQube, for example, is proprietary, so although it deploys easily it lacks modification and experimentation support. Furthermore, most of the academic quality modeling frameworks are no longer supported and have dependency and source control issues. Thus, a platform that provides common ground for deployment strategies along with modern support is a valuable contribution.

Model Evaluation Exposure

Question Q01-04, *do the models produced facilitate ease of evaluation by researchers*, is asked to analyze the process of model generation with respect to experimental opportunity. The question is asked from the point of view of a quality researcher who has run quality assessment and desires to use the quality assessment output to understand more about the state of the system under evaluation. For example, in the most rudimentary form of quality assessment output, a researcher may learn that the total quality state of the system changed from 0.87 to 0.76. A

quality model output that provides more opportunity for experimental evaluation would additionally provide the values of each quality model factor node beyond just the total quality index. A quality model that provides even further opportunity for evaluation would not provide just the quality values of each quality node, but also show the values of every weighted edge, the field values of relevant internal objects, and metadata about the system under evaluation.

Q01-04 Interpretation Algorithm The perspective taken in this thesis regarding improvement of experimental opportunity is that more information output by model assessment suggests better experimental opportunity. Of the models reviewed in chapters 2 and 3, proprietary quality assessment frameworks simply returned either the total quality index value or the quality values of the quality aspect nodes such as total quality and maintainability. Of the remaining models reviewed that were still supported and able to be operationalized, Quamoco showed an exemplary performance in its output by providing a *.xml* file of every edge and node value. This output, however, is a massive *.xml* file, difficult to interpret, and was not designed to facilitate experimental evaluation in its raw form. At best, the other supported models reviewed did not output more than their node values and edge weights.

Table 7.4: Q01-04 metric

Metric	Description	Value
M17	Model output exposure.	0.81

The relevant metric M17, as shown in table 7.4, represents a ratio of the total possible data fields an assessment output file could represent compared to the data fields actually output by the assessment. In the context of a PIQUE-generated quality

model, table 7.5 shows the total possible fields that could be represented resulting from the test case of section 6.5.

Table 7.5: PIQUE output exposure test

Row	Internal Mechanisms, Objects, Values	Exposed?
1	Static analysis tool objects	
2	Static analysis tool names	✓
3	Benchmarking strategy name	✓
4	Benchmarking strategy logic	
5	Benchmarking output data	✓
6	Weighting strategy name	✓
7	Weighting strategy logic	
8	Weighting output data	✓
9	Quality model	✓
10	Model node name	✓
11	Model node final value	✓
12	Model node evaluation strategy name	✓
13	Model node evaluation strategy logic	
14	TQI node	✓
15	TQI incoming weights	✓
16	Quality Aspect nodes	✓
17	Quality Aspect incoming weights	✓
18	Product Factor nodes	✓
19	Product Factor incoming weights	✓
20	Measure nodes	✓
21	Measure node +/− impact	✓

22	Measure node normalizer	✓
23	Measure node incoming Diagnostics	✓
24	Measure node utility function thresholds	✓
25	Measure node value before normalization	
26	Measure node value before utility function	
27	Diagnostic nodes	✓
28	Diagnostic associated tool name	✓
29	Diagnostic incoming Findings	✓
30	Finding nodes	✓
31	Finding file path	✓
32	Finding line number	✓
33	Finding character number	✓
34	Finding numerical value	✓
35	Finding additional info (e.g. severity)	✓
36	Normalizer object	
37	Benchmarker object	
38	Evaluator object	
39	Project	✓
40	Project name	✓
41	Project lines of code	✓
42	Project path	✓
43	Project's quality model	✓

The interpretation algorithm of listing 7.4 returns improvement of experimental opportunity if the input quality model produces assessment results with more information than just the model and node names, node values, and edge weights.

Of the rows in table 7.5, 14 rows relate to model and node names, node values, and edge weights. Thus, initial improvement is realized if $M16 > \frac{14}{43}$. Of the rows contributing to the numerator value, 14 of them must be from the rows relating to model and node names, node values, and edge weights in order to meet the same type of exposure represented by the model frameworks used in comparison. The resulting ratio of M16 from testing is $M16 = exposure = \frac{exposed}{total} = \frac{35}{43} = 0.81$.

Listing 7.4: Q01-04 interpretation algorithm

```
target = 14/43
repeat until best possible target achieved
    if rows 10-11, 14-21, 27, 29-30, and 24
        are not exposed then
            increase exposure of assessment output
    if M16
        > target then
            method better than history
            target = M16
        <= target then
            increase exposure of assessment output
```

Q01-04 Discussion Because PIQUE exposes node values, edge weight values, and much more, the interpretation algorithm shows improvement is achieved. Unlike the other model frameworks reviewed, PIQUE is designed with an easily consumable output file in mind. Proprietary approaches conceal most model values and approaches like Quamoco generate a confusing *.xml* file while PIQUE outputs tidy *.json* files, exposing just key-value pairs of data that would be beneficial to experimentative assessment.

There is a line between not enough information, and too much, so a perfect exposure value may not be 1.0 (43 fields exposed out of 43). In fact, in the context of table 7.5, exposing data for objects such as the static analysis tools or a normalizer object may not be desired given their fields are mostly relevant to internal Java object construction designs.

Model Validity

Question Q01-05, *are the models produced valid*, is asked to analyze the process of quality model validation. A requirement for any goal relating to a quality model is that the model itself be valid: the quality assessment scores decrease when poor quality is introduced, and the scores increase when improvement is introduced. To assess validity, *in vitro* tests using PIQUE-generated models are run[2] resulting in the metric data of table 7.6

Table 7.6: Q01-05 metrics

Metric	Description	Value
M13	The change in quality assessment score decreases after introducing flaws	True
M14	The change in quality assessment score increases after introducing improvements	True
M15	The change in quality assessment score after modifying subjective quality opinion reflects the new prioritization	True

Given the necessity for the models produced by PIQUE to be valid in order to be used to analyze quality model processes, a satisfying evaluation of the boolean

[2]Section 6.6.

Q01-05 metrics is,

$$M13 \wedge M14 \wedge M15 = True$$

Filling in these values using table 7.6 reveals a the model under test, as generated by PIQUE, is valid. The boolean equation must evaluate to true before considering the other metrics and questions in support of the GQM goals. PIQUE-produced models must work on a fundamental level as a base line of improvement and validity.

Goal 01 Conclusions

Ultimately, did improvement with respect to effort investment, experimentation, and collaborative opportunity from the point of view of quality model researchers regarding the process of generating, validating, and operationalizing quality models occur?

Using PIQUE, the primary blockers for researchers are identifying static analysis tools for their language of choice, learning how to run their analysis, and collecting a satisfying collection of benchmark repositories. However, these blockers are inevitable given the platform is language agnostic and there is no way to support all unknown, future tools and languages. Apart from these blockers, a new researcher does not need to learn and implement difficult derivation mechanisms nor invest time into defining a derivation or assessment runnable process since the platform provides these necessary elements by default.

As a practitioner becomes more familiar with the domain of quality modeling, the platform provides interfaces to introduce exploratory ideas without needing to modify the rest of the system. This can be accomplished in a matter of hours. Additionally, PIQUE and its current actualized extensions such as PIQUE-CSHARP exist as open source projects with modern dependency and deployment management capabilities.

Of the exercises run to generate metrics relevant to the questions of goal G01, the PIQUE-generated model did not show improvement with regards to operationalization effort; however, given the evidence of improvement regarding model generation effort, model mechanism modification effort, experimental opportunity, and validity of evaluation, the goal is considered achieved.

7.2 Goal 02: The Industry Perspective

Using the goal/question/metric (GQM) of [6], goal two states,

> "Analyze a process of generating and operationalizing quality models for the purpose of improvement with respect to cost investment and acceptability from the point of view of software development practitioners in the context of static software system analysis."

The following sections answer the questions relevant to goal G02 using simple interpretation algorithms. Finally, the the results obtained for G02 are discussed using the evaluations of questions Q02-01 through Q02-03.

Model Tuning Expense

Question Q02-01, *is it expensive to tune a model to a company's need*, is asked to analyze the process of model generation with respect to the cost of time investment. A scenario relating to this question is that of an industry practitioner working with a quality modeling expert to tune a model to their subjective needs.

Quality models need to represent the inherent subjectivity of a stakeholder's views to correctly value the aspects of quality that matter to their needs. To accomplish this, PIQUE supports a linguistic method of enacting the analytical hierarchy process to be used as input to the automated model derivation process.

An exercise to record the time-cost necessary for a practitioner to understand the basic quality concepts and record their subjective declarations was carried out.[3] The exercise was used to produce a value for metric M12 as shown in table 7.7.

Table 7.7: Q02-01 metrics

Metric	Description	Value
M12	Time taken by a practitioner to express subjective quality definitions.	2 hours

Q02-01 Interpretation Algorithm Because no data is available to compare metric M12 values to other quality modeling efforts using methods similar to AHP, a practitioner from an industry partner was interviewed regarding his opinion of the largest value M12 could be before the efforts were no longer worth the time-cost. A value of $M12 \leq 4$ hours was given. This value is used as the basis for improvement of the interpretation algorithm of listing 7.5. The algorithm states that improvement is first accomplished if the AHP takes less than four hours.

Listing 7.5: Q02-01 interpretation algorithm

```
1 target = 4 hours
2 repeat until best possible target achieved
3   if M12
4     >= target then
5       lower AHP process time
6     < target then
7       method better than history
8       target = M12
```

[3]Section 6.4.

Discussion Metric M12 is less than the target, so improvement is realized; however, the model used was relatively small and only the quality aspect to total quality index matrix was filled in, so M12 comes from a relatively simple case. In more complex scenarios involving multiple layers of pairwise comparisons, the value of M12 will grow exponentially. This scenario will likely occur for a security model, so other weighting algorithms should be considered; fortunately, PIQUE is designed to support such modification. Additionally, given the basis of improvement value comes from interviewing a single practitioner, the output of the improvement algorithm contains a threat to external validity; however, the conclusions of this chapter do not attempt to claim statistical significance, but rather provide data for metrics to answer questions relating to goals of improvement.

Assessment Integration Acceptability

Question Q02-02, *how acceptable is it to integrate quality model assessment into an external, continuous integration system*, is asked to analyze the process of model operationalization with respect to acceptability from the context of industry practitioners.

An exercise is conducted that deploys an operationalized PIQUE-CSHARP quality assessment engine into an industry partner's software continuous integration system.[4] After discussion, it was concluded that the main concerns of integration of an assessment engine into a pre-existing environment are the number of additional dependencies that would need to be installed and whether the process would interfere with the normal development continuous integration pipeline. These acceptability

[4]Section 6.4.

concepts are represented as metrics M17 and M18 as shown in table 7.8

Table 7.8: Q02-02 metrics

Metric	Description	Value
M17	Number of external dependencies needed for actualization	1
M18	[True | False] Failure of assessment does not interfere with other system processes	True

Regarding the practitioner's interpretation of acceptable values, no additional dependencies (apart from the *.jar* assessment engine itself) was expressed as preferred. It was strongly worded that assessment can also not interfere with other CI pipeline processes. Regarding historical academic quality modeling efforts, only one system was found that specifically integrated in to a practitioner's continuous integration [19], and the fewest external dependencies possible from an operating system independent perspective is one. So, improvement is achieved if, given an assessment engine deployed into a continuous integration environment, the number of external dependencies equals one and the assessment process does not interfere with native processes.

$$(|M17| \leq 1) \wedge (M18 = True)$$

The tests producing M17 and M18 satisfy the equation.

Trust in Assessment Values

Question Q02-03, *can quality models be used such that their output values are trusted by practitioners*, is asked to analyze the process of quality model operationalization with respect to output acceptability. To evaluate this, an exercise is

run that integrates PIQUE-CSHARP into an industry partner's software development life cycle.[5] The partner is asked to provide sequences of commit histories where it was believed moments of quality improvement occurred. The quality engine then runs assessment of the first and last commits of each given historical sequence. If there is a perfect matching of the practitioner's opinion of quality increase and the quality assessment results of quality increase, metric M19 is marked as true. The commit histories, assessment values, and M19 results are given in tables 7.9 and 7.10.

Table 7.9: Commit histories of subjectively perceived improvement. S-Quality: Starting Quality. F-Quality: Finish Quality.

Start	*Finish*	*# Commits*	*S-Quality*	*F-Quality*	*Pos. slope?*
Oct. 22	Dec. 04	157	0.4335	0.4102	
Jan. 21	Feb. 10	171	0.4087	0.4186	✓
Jan. 30	Apr. 15	608	0.4082	0.3840	
Mar. 25	Mar. 27	26	0.3826	0.3836	✓

Table 7.10: Question 02-03 metric

Metric	Description	Value
M19:	[True | False] The slope of assessed values over time matches practitioner opinion.	False

Discussion The results of table 7.9 can be interpreted in more than one way. One can assume the industry practitioner's opinion that objective quality did increase

[5]Section 6.6.

during the given commit history. If this is the case, it suggests the C# PIQUE quality model was not sensitive enough to refactors and quality improvements.

One could also interpret the results from the perspective that, although improvements were introduced, more cases of poor quality were also introduced. This scenario could explain why the shorter histories of improvement (January 21-February 10 and March 25-March 27) had positive improvement realized from both practitioner opinion and quality assessment output, but phases of longer commit histories (October 22-December 04 and January 30-April 15) were in conflict. This rationale is sensible: apart from major refactoring efforts, as a system grows larger it will likely lower in quality because there are more areas for poor quality findings.

A final interpretation could be that the practitioner's opinion of quality improvement was incorrect. Evaluating the true change in quality would require intense evaluation from industry experts, a task outside the scope of this work.

In summary, the test cases associated with answering question Q02-03 are performed to give an initial intuition if a PIQUE-derived quality model can output values aligned with subjective opinion of quality changes. While the results of quality changes over short periods of time are encouraging, the evidence thus far remains inconclusive.

Goal 02 Conclusions

Although these results represent early evaluations, they reveal a partial achievement of goal 02. The default mechanisms of PIQUE for tuning a model to a company's need (as inspired by QATCH [37]) shows effective improvement regarding stakeholder cost compared to other approaches, mostly because other approaches do not feature a way for stakeholders to express subjectivity without requirement of expertise in quality modeling. Of approaches that do feature subjective injection, their method

of expression uses numeric values rather than the linguistic values used by PIQUE.

In the context of the test cases, success is realized regarding the acceptability of model assessment into external, continuous integration systems. This success is positive evidence toward the achievement of goal 02, most notably with respect to practitioner acceptability and trustability. In conversations with practitioners, there was much interest and agreement expressed regarding viewing quality as a "delta-Q" that is only applicable to a single system's changes over time. If quality is asserted as an isolated value without context, practitioners expressed distrust and uncertainty regarding the meaning of the value. On the other hand, the expression of quality changing compared to the value it was perviously evaluated to was intuitive and trustable. In order to accommodate persistent quality assessments as a software product evolves, integration of the assessment engine into a CI environment is necessary.

The exercise relating to question Q02-03 shows that, in the case of the PIQUE-derived ISO 25010 C# model of appendix G, quality assessment results over commit histories will sometimes agree with a practitioner's opinion of quality changes, but some times the assessment results will not agree. This result can still support the improvements desired by goal 02 by expressing agreement with practitioners in simple cases, and opens a channel of communication regarding what is occurring to the product's quality in more complex cases.

Research of model output trust and verification is difficult, requiring more investigation. PIQUE is designed to facilitate such research and can provide a platform to further progress in this field towards a goal of more widespread practitioner acceptance.

CHAPTER EIGHT

THREATS TO VALIDITY

Wohlin et al. [45] categories of threats to validity in software research. Internal threats to validity refer to undesired relationships, and the extent to which independent variables cause effects on a dependent variable. External threats to validity apply to the degree that findings of the study can be generalized to other environments. Construct threats to validity refer to how representative the study's measures represent their intended real-world constructs.

Given that the result data comes from test cases (chapter 6) rather than statistically driven experimentation, there exists a persistent threat to validity across all results. This concern is mitigated due to the purpose of this thesis to provide the groundwork necessary for future experimentation; that is, the results do not make statistical claims but instead are presenting evidence regarding the nature of improvement. As it stands, the quality models compared at a high level are too different to have a level of acceptable internal, external, and conclusion validity. In the future, models generated by PIQUE will have acceptable levels of comparability. Providing this opportunity is a primary contribution of this thesis.

8.1 Internal Validity

All presented interpretation algorithms that compare a PIQUE model to a historical model from chapter 2 or 3 (such as QATCH or Quamoco) bring a threat to internal validity because the two models compared are so different that the metrics used for improvement cannot be considered dependent variables. For any presented improvement algorithm in chapter 7, there are likely numerous other unaccounted for

reasons why improvement was achieved or not achieved. This concern is mitigated by the recognition that models, in the current state of the quality modeling research domain, are not comparable, so the exercises of this thesis make weak assertions of improvement while providing a way forward to make valid comparisons in the future using the platform PIQUE.

The effort metrics used also bring concerns. As efforts, such as time, were being generated, the researcher under study was also the researcher who designed the system. Thus, bias and misrepresentation of expected effort likely occurred. This is mitigated somewhat due to the nature of addressing improvement with regard to effort. Even if the effort results are biased to under-represent, the difference is still on the scale of hours or days versus months or years. Given the interpretation approach used for the assessments, it is thus not possible to assert any causal inferences from the data.

8.2 External Validity

The result of the metric regarding model output trust, M19, and effort metric M12 of practitioner time taken to fill in subjective quality definitions represent threats to external validity. Before running the relevant test cases, a significant amount of time had already been spent with the industry partner. During those times, many conversations were had regarding quality modeling, so the practitioner was primed with a priori knowledge before providing effort and opinion results.

Any test case that used the PIQUE-CSHARP control model has an inherent threat to external validity. The results can only apply to similar model designs using the same language, the same static analysis tools, a similar benchmark repository, and similar mechanisms. For any interpretation algorithm expressing improvement, improvement can only be claimed for the specific model generated by PIQUE versus

the specific other modeling approach used in comparison. Clearly, generalization of the results cannot be claimed. Instead, the test cases and discussions look to present examples that PIQUE is capable of producing improved quality models in contrast to similar, other recent quality modeling attempts. Finally, the industry practitioner feedback regarding trust and acceptability can not generalize outside the domain of the industry partner interacted with.

8.3 Construct Validity

Regarding the ISO/IEC C# quality model derived by PIQUE, a design choice is made to measure all product factors with a single *"FactorName_Smells"* measure. These measures are designed by searching all available diagnostics provided by the static analysis tools used that represent problematic code and connecting it to the appropriate product factor "smell" measure. Thus, the measurement of low level factor concepts is accomplished strictly through a count-based representation of possible problems. This can lead to concerns such as under-representation of a given measure due to lack of tool support or a misclassification of a diagnostic to its correct measure. This is mitigated somewhat through use of a benchmark repository, but the measurements used in the quality models during the tests may still not represent their intended construct well. The main affect of this threat is on the trustability of the model. Other test cases are not as dependent on the validity of the measurement constructs used.

The body of measures used as a whole is also a threat. Given it is not determined how many measures (or few) are needed to quantify their associated product factors and the representative strength of the measure is not addressed, this threat to construct validity impacts the representative meaning of the factor nodes. Evaluating the change in factor quality values rather than the meaning of a singular value can

mitigate this threat.

Finally, the effort metrics used also carry construct validity concerns. The values of man-hours in the test cases come from the tests being carried out by one participant; however, man-hours does not function as a linear scalar thus it may not generalize to represent effort when larger number of participants are involved.

CHAPTER NINE

CONCLUSION

This thesis presents a platform intended to facilitate the processes of software quality modeling: the derivation of new models using new approaches, the validation of the model's output, and the operationalization of the model into real-world systems. To evaluate the capabilities of such a platform, test cases are run on the models and operations produced by the platform pertaining to academic and industrial goals. The first goal is to analyze the processes involved in generating, validating, and operationalizing quality models for the purpose of improvement with respect to effort investment, experimentation, and collaborative opportunity from the point of view of quality model researchers in the context of static software system analysis. The second goal analyzes the same processes but from the point of view of software development practitioners regarding feasibility and acceptability.

The platform used to generate the quality models under evaluation, PIQUE, is designed to handle the language-agnostic aspects of quality modeling and assessment while leaving the necessary language-specific components under the responsibility of the quality modeler. The platform provides a default state of quality assessment mechanisms, but all involved mechanisms are extensible through a collection of libraries and interfaces.

To address the research goals, PIQUE is used to generate an ISO/IEC 25010 based quality model operationalized for C# .NET systems. The results provide good evidence that the platform can offer major improvements to quality operations by reducing the overhead needed to design and derive quality models, experiment with new design mechanisms, operationalize into new languages and systems, and validate

results.

A PIQUE-generated model is introduced to an industrial system where the capability to inject subjective quality opinions in an automated fashion is investigated. While the approach was found to be low-cost and easy to deploy in foreign development environments, there still remains general hesitation about the representative strength of quality model assessed values–a pervasive problem in quality modeling history.

Finally, a security-focused model is designed using the constructs offered by PIQUE to demonstrate capability to derive niche and experimental models. The model uses the ISO/IEC 25010 decomposition of security along with a security-focused architectural hierarchy provided by MITRE's Common Weakness Enumeration (CWE). The exercise of making a C# security-focused quality model revealed there is currently not enough open source tool support to gather an acceptably sized base of potential findings in order for the ISO/IEC/CWE security assessment to evaluate with meaningful information. This reveals a need for further research and development in security-focused static analysis tools across a wider breadth of programming languages.

While many of questions relating to the goal/question/metric paradigm used in this thesis are met, there is much more research and validation needed, and PIQUE is designed to support further quality modeling research. Specifically, quality assessment and understanding in the domain of security is in high need of further academic research and solutions. The tools used in the test cases of this thesis are strictly static analysis tools, but dynamic assessment can also be introduced. Finally, given each component in the platform's model derivation and quality assessment engines is designed with modularity and extensibility in mind, mechanism experimentation on components such as evaluation strategies, utility function approaches, and new ways

of injecting subjective quality awareness can and should be enacted.

REFERENCES CITED

[1] H. Al-Kilidar, K. Cox, and B. Kitchenham. The use and usefulness of the iso/iec 9126 quality standard. In *2005 International Symposium on Empirical Software Engineering, 2005.*, pages 7–pp. IEEE, 2005.

[2] T. Alves, J. Correia, and J. Visser. Benchmark-based aggregation of metrics to ratings. *Proc. IWSM/Mensura*, pages 20–29, 2011.

[3] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman. Managing technical debt in software engineering (dagstuhl seminar 16162). In *Dagstuhl Reports*, volume 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[4] J. Bansiya and C. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Software Eng.*, 28:4–17, 2002.

[5] F. H. Barron and B. E. Barrett. Decision quality using ranked attribute weights. *Management science*, 42(11):1515–1523, 1996.

[6] V. R. Basili. Software modeling and measurement: the goal/question/metric paradigm. Technical report, 1992.

[7] B. Boehm. *Characteristics of software quality*. TRW series of software technology. North-Holland Pub. Co., 1978.

[8] W. Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1992.

[9] M. R. Dale and C. Izurieta. Impacts of design pattern decay on system quality. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–4, 2014.

[10] F. Deissenboeck, L. Heinemann, M. Herrmannsdoerfer, K. Lochmann, and S. Wagner. The quamoco tool chain for quality modeling and assessment. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 1007–1009. IEEE, 2011.

[11] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y. Parareda, and M. Pizka. Tool support for continuous quality control. *IEEE Software*, 25(5):60–67, 2008.

[12] F. Deissenboeck, E. Juergens, K. Lochmann, and S. Wagner. Software quality models: Purposes, usage scenarios and requirements. pages 9–14, Vancouver, BC, 2009. ICSE Workshop on Software Quality, IEEE.

[13] R. G. Dromey. A model for software product quality. *IEEE Transactions on Software Engineering*, 21(2):146–162, 2 1995.

[14] I. O. for Standardization/International Electrotechnical Commission et al. Iso/iec 25010—systems and software engineering—systems and software quality requirements and evaluation (square)—system and software quality models. *Authors, Switzerland*, 2011.

[15] X. Franch and J. Carvallo. Using quality models in software package selection. *Software, IEEE*, 20:34–41, 02 2003.

[16] R. B. Grady and D. L. Caswell. *Software metrics: establishing a company-wide program*. Prentice-Hall, Inc., 1987.

[17] I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In *6th international conference on the quality of information and communications technology (QUATIC 2007)*, pages 30–39. IEEE, 2007.

[18] A. Hunt, D. Thomas, and W. Cunningham. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, 2000.

[19] C. Izurieta, I. Griffith, and C. Huvaere. An industry perspective to comparing the sqale and quamoco software quality models. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 287–296. IEEE, 2017.

[20] C. Izurieta, I. Griffith, D. Reimanis, and R. Luhr. On the uncertainty of technical debt measurements. In *2013 International Conference on Information Science and Applications (ICISA)*, pages 1–4. IEEE, 2013.

[21] C. Izurieta, I. Ozkaya, C. B. Seaman, P. Kruchten, R. L. Nord, W. Snipes, and P. Avgeriou. Perspectives on managing technical debt: A transition point and roadmap from dagstuhl. In *QuASoQ/TDA APSEC*, pages 84–87, 2016.

[22] C. Izurieta and M. Prouty. Leveraging secdevops to tackle the technical debt associated with cybersecurity attack tactics. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 33–37. IEEE, 2019.

[23] C. Izurieta, G. Rojas, and I. Griffith. Preemptive management of model driven technical debt for improving software quality. In *2015 11th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*, pages 31–36. IEEE, 2015.

[24] C. Izurieta, A. Vetrò, N. Zazworka, Y. Cai, C. Seaman, and F. Shull. Organizing the technical debt landscape. In *2012 Third International Workshop on Managing Technical Debt (MTD)*, pages 23–26. IEEE, 2012.

[25] B. Kitchenhamm, S. G. Linkman, A. Pasquini, and V. Nanni. The squid approach to defining a quality model. *Software Quality Journal*, 6:211–233, 1997.

[26] M. Kläs, C. Lampasona, S. Nunnenmacher, S. Wagner, M. Herrmannsdörfer, and K. Lochmann. How to evaluate meta-models for software quality. In *Proceedings of the 20th International Workshop on Software Measurement (IWSM2010)*, 2010.

[27] J.-L. Letouzey and T. Coq. The sqale analysis model: An analysis model compliant with the representation condition for assessing the quality of software source code. In *2010 Second International Conference on Advances in System Testing and Validation Lifecycle*, pages 43–48. IEEE, 2010.

[28] N. Leveson. The role of software in spacecraft accidents. *AIAA Journal of Spacecraft and Rockets*, 2004.

[29] K. Lochmann. A benchmarking-inspired approach to determine threshold values for metrics. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–8, 2012.

[30] M. Mantyla, J. Vanhanen, and C. Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 381–384. IEEE, 2003.

[31] K. Mordal-Manet, F. Balmas, S. Denier, S. Ducasse, H. Wertz, J. Laval, F. Bellingard, and P. Vaillergues. The squale model—a practice-based industrial quality model. In *2009 IEEE International Conference on Software Maintenance*, pages 531–534. IEEE, 2009.

[32] B. Norick, J. Krohn, E. Howard, B. Welna, and C. Izurieta. Effects of the number of developers on code quality in open source software: a case study. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–1, 2010.

[33] D. Reimanis, C. Izurieta, R. Luhr, L. Xiao, Y. Cai, and G. Rudy. A replication case study to measure the architectural quality of a commercial system. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–8, 2014.

[34] T. L. Saaty. Decision making with the analytic hierarchy process. *International journal of services sciences*, 1(1):83–98, 2008.

[35] I. Samoladas, G. Gousios, D. Spinellis, and I. Stamelos. The sqo-oss quality model: Measurement based open source software evaluation. In B. Russo, E. Damiani, S. Hissam, B. Lundell, and G. Succi, editors, *Open Source Development, Communities and Quality*, pages 237–248, Boston, MA, 2008. Springer US.

[36] J. C. Santos, K. Tarrit, and M. Mirakhorli. A catalog of security architecture weaknesses. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 220–223. IEEE, 2017.

[37] M. Siavvas, K. Chatzidimitriou, and A. Symeonidis. Qatch - an adaptive framework for software product quality assessment. *Expert Systems With Applications*, 86:350–366, 2017.

[38] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM systems journal*, 13(2):115–139, 1974.

[39] R. Van Zeist and P. Hendriks. Specifying software quality with the extended iso model. *Software Quality Journal*, 5:273–284, 1996.

[40] S. Wagner, A. Goeb, L. Heinemann, M. Kläs, C. Lampasona, K. Lochmann, A. Mayr, R. Plösch, A. Seidl, J. Streit, et al. Operationalised product quality models and assessment: The quamoco approach. *Information and Software Technology*, 62:101–123, 2015.

[41] S. Wagner, K. Lochmann, S. Winter, F. Deissenboeck, E. Juergens, M. Herrmannsdoerfer, L. Heinemann, M. Kläs, J. Heidrich, R. Ploesch, A. Göeb, and C. Koerner. The quamoco quality meta-model, technical report. Technical Report TUM-I1281, Technische Universität, München, 2012.

[42] S. Wagner, K. Lochmann, S. Winter, A. Göeb, and M. Kläs. Quality models in practice: A preliminary analysis. In *ESEM 2009*, 2009.

[43] S. Wagner, K. Lochmann, S. Winter, A. Göeb, M. Kläs, and S. Nunnenmacher. Software quality in practice. survey results. Technical Report Technical Report TUM-I129, Technische Universität München, 2012.

[44] C. Williams. Anatomy of openssl's heartbleed: Just four bytes trigger horror bug. `https://www.theregister.co.uk/2014/04/09/heartbleed_explained`, Apr 2014. Accessed on 2020-02-20.

[45] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

APPENDICES

APPENDIX A

C# QUALITY MODEL DESCRIPTION

```
{
  "name": "CSharp ISO25k Quality Model",
  "additionalData": {},
  "factors": {
    "tqi": {
      "Total Software Quality": {
        "description": "The total quality value representing the entire system"
      }
    },
    "quality_aspects": {
      "Compatibility": {
        "description": "Degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions while sharing the same hardware or software environment"
      },
      "Maintainability": {
        "description": "This characteristic represents the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements"
      },
      "Performance Efficiency": {
        "description": "This characteristic represents the performance relative to the amount of resources used under stated conditions"
      },
      "Portability": {
        "description": "Degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another"
      },
      "Reliability": {
        "description": "Degree to which a system, product or component performs specified functions under specified conditions for a specified period of time"
      },
      "Security": {
        "description": "Degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization"
      },
```

```
29        "Usability": {
30          "description": "Degree to which a product or system can be used by
              ↪  specified users to achieve specified goals with effectiveness,
              ↪  efficiency and satisfaction in a specified context of use"
31        }
32      },
33      "product_factors": {
34        "Data Type Integrity": {
35          "description": "A  representation of how well each object adheres
              ↪  to CSharp object oriented typing expectations"
36        },
37        "Documentation": {
38          "description": "Information about the source code intended to
              ↪  assist human understanding"
39        },
40        "Encryption": {
41          "description": "The concealment of code elements such that external
              ↪  entities are unable to decipher its information"
42        },
43        "Exception Handling": {
44          "description": "Goodness of approach in handling exceptions in a
              ↪  compliant and productive way"
45        },
46        "Functional Syntax": {
47          "description": "The operators, syntactic decision, object
              ↪  selection, etc., that determine how a functional act is
              ↪  defined. This is disjoint from the Format property by having
              ↪  potential impact on more than just human-readability"
48        },
49        "Format": {
50          "description": "Code formatting. This involves how the code is
              ↪  written as it appears to a human reader. Common synonyms
              ↪  include Style, Simplification, Readability, Convention,
              ↪  Verbosity, Naming"
51        },
52        "IO Handling": {
53          "description": "Management of the data going into and out of a
              ↪  source component.  Can relate to data sanitization, filtering,
              ↪  rejection, querying, database queries and storage, etc"
54        },
55        "Resource Handling": {
56          "description": "Management of resources: often related to locking
              ↪  of instances and file IO"
57        },
58        "Structure": {
```

```
59          "description": "The organization of classes, files, and methods.
              ↪  Includes object oriented concepts such as inheritance"
60        }
61      }
62    },
63    "measures": {
64      "DTI Smells": {
65        "description": "Code smells related to data type integrity",
66        "positive": false,
67        "parents": [
68          "Data Type Integrity"
69        ],
70        "diagnostics": [
71          {
72            "name": "loc",
73            "description": "Normalizer diagnostic lines of code",
74            "eval_strategy": "pique.evaluation.LOCDiagnosticEvaluator",
75            "toolName": "RoslynatorLoc"
76          },
77          {
78            "name": "RCS1008",
79            "description": "Use explicit type instead of 'var' (when the type
              ↪  is not obvious)",
80            "toolName": "Roslynator"
81          },
82          {
83            "name": "RCS1009",
84            "description": "Use explicit type instead of 'var' (foreach
              ↪  variable)",
85            "toolName": "Roslynator"
86                  },
87                  // ...
88          {
89            "name": "RCS1234",
90            "description": "Duplicate enum value",
91            "toolName": "Roslynator"
92          }
93        ]
94      },
95      "Documentation Smells": {
96        "description": "Code smells related to documentation",
97        "positive": false,
98        "parents": [
99          "Documentation"
100       ],
```

```
      "diagnostics": [
        {
          "name": "loc",
          "description": "Normalizer diagnostic lines of code",
          "eval_strategy": "pique.evaluation.LOCDiagnosticEvaluator",
          "toolName": "RoslynatorLoc"
        },
        {
          "name": "RCS1138",
          "description": "Add summary to documentation comment",
          "toolName": "Roslynator"
        },
        {
          "name": "RCS1139",
          "description": "Add summary element to documentation comment",
          "toolName": "Roslynator"
        },
        // ...
        {
          "name": "RCS1232",
          "description": "Order elements in documentation comment",
          "toolName": "Roslynator"
        }
      ]
    },
    "Encryption Smells": {
      "description": "Code smells related to Encryption",
      "positive": false,
      "parents": [
        "Encryption"
      ],
      "diagnostics": [
        {
          "name": "loc",
          "description": "Normalizer diagnostic lines of code",
          "eval_strategy": "pique.evaluation.LOCDiagnosticEvaluator",
          "toolName": "RoslynatorLoc"
        },
        {
          "name": "SCS0005",
          "description": "Weak Random Number Generator",
          "toolName": "Roslynator"
        },
        {
          "name": "SCS0006",
```

```
146             "description": "Weak hashing function",
147             "toolName": "Roslynator"
148           },
149           // ...
150           {
151             "name": "SCS0034",
152             "description": "Password RequiredLength Not Set",
153             "toolName": "Roslynator"
154           }
155         ]
156       },
157       "Exception Handling Smells": {
158         "description": "Code smells related to exception handling",
159         "positive": false,
160         "parents": [
161           "Exception Handling"
162         ],
163         "diagnostics": [
164           {
165             "name": "loc",
166             "description": "Normalizer diagnostic lines of code",
167             "eval_strategy": "pique.evaluation.LOCDiagnosticEvaluator",
168             "toolName": "RoslynatorLoc"
169           },
170           {
171             "name": "RCS1044",
172             "description": "Remove original exception from throw statement",
173             "toolName": "Roslynator"
174           },
175           {
176             "name": "RCS1075",
177             "description": "Avoid empty catch clause that catches
                ↪  System.Exception",
178             "toolName": "Roslynator"
179           },
180           // ...
181           {
182             "name": "RCS1236",
183             "description": "Use exception filter",
184             "toolName": "Roslynator"
185           }
186         ]
187       },
188       "Functional Syntax Smells": {
189         "description": "Code smells related to functional syntax",
```

```
      "positive": false,
      "parents": [
        "Functional Syntax"
      ],
      "diagnostics": [
        {
          "name": "loc",
          "description": "Normalizer diagnostic lines of code",
          "eval_strategy": "pique.evaluation.LOCDiagnosticEvaluator",
          "toolName": "RoslynatorLoc"
        },
        {
          "name": "RCS1048",
          "description": "Use lambda expression instead of anonymous
          ↪  method",
          "toolName": "Roslynator"
        },
        {
          "name": "RCS1061",
          "description": "Merge if statement with nested if statement",
          "toolName": "Roslynator"
        },
        // ...
        {
          "name": "RCS1236",
          "description": "Use exception filter",
          "toolName": "Roslynator"
        }
      ]
    },
    "Format Smells": {
      "description": "Code smells related to format",
      "positive": false,
      "parents": [
        "Format"
      ],
      "diagnostics": [
        {
          "name": "loc",
          "description": "Normalizer diagnostic lines of code",
          "eval_strategy": "pique.evaluation.LOCDiagnosticEvaluator",
          "toolName": "RoslynatorLoc"
        },
        {
          "name": "RCS1001",
```

```
234             "description": "Add braces (when expression spans over multiple
                ↪  lines)",
235             "toolName": "Roslynator"
236           },
237           {
238             "name": "RCS1002",
239             "description": "Remove braces",
240             "toolName": "Roslynator"
241           },
242           // ...
243           {
244             "name": "RCS1215",
245             "description": "Expression is always equal to true/false",
246             "toolName": "Roslynator"
247           }
248         ]
249       },
250       "IO Handling Smells": {
251         "description": "Code smells related to IO handling",
252         "positive": false,
253         "parents": [
254           "IO Handling"
255         ],
256         "diagnostics": [
257           {
258             "name": "loc",
259             "description": "Normalizer diagnostic lines of code",
260             "eval_strategy": "pique.evaluation.LOCDiagnosticEvaluator",
261             "toolName": "RoslynatorLoc"
262           },
263           {
264             "name": "SCS0001",
265             "description": "Command Injection",
266             "toolName": "Roslynator"
267           },
268           {
269             "name": "SCS0002",
270             "description": "SQL Injection (LINQ)",
271             "toolName": "Roslynator"
272           },
273           // ...
274           {
275             "name": "SCS0036",
276             "description": "SQL Injection (EnterpriseLibrary.Data)",
277             "toolName": "Roslynator"
```

```
278           }
279         ]
280       },
281       "Resource Handling Smells": {
282         "description": "Code smells related to resource handling",
283         "positive": false,
284         "parents": [
285           "Resource Handling"
286         ],
287         "diagnostics": [
288           {
289             "name": "loc",
290             "description": "Normalizer diagnostic lines of code",
291             "eval_strategy": "pique.evaluation.LOCDiagnosticEvaluator",
292             "toolName": "RoslynatorLoc"
293           },
294           {
295             "name": "RCS1059",
296             "description": "Avoid locking on publicly accessible instance",
297             "toolName": "Roslynator"
298           },
299           {
300             "name": "RCS1090",
301             "description": "Call 'ConfigureAwait(false)'",
302             "toolName": "Roslynator"
303           },
304           // ...
305           {
306             "name": "VSTHRD200",
307             "description": "Use Async naming convention",
308             "toolName": "Roslynator"
309           }
310         ]
311       },
312       "Structure Smells": {
313         "description": "Code smells related to structure",
314         "positive": false,
315         "parents": [
316           "Structure"
317         ],
318         "diagnostics": [
319           {
320             "name": "loc",
321             "description": "Normalizer diagnostic lines of code",
322             "eval_strategy": "pique.evaluation.LOCDiagnosticEvaluator",
```

```
            "toolName": "RoslynatorLoc"
          },
          {
            "name": "RCS1060",
            "description": "Declare each type in separate file",
            "toolName": "Roslynator"
          },
          {
            "name": "RCS1085",
            "description": "Use auto-implemented property",
            "toolName": "Roslynator"
          },
          // ...
          {
            "name": "RCS1241",
            "description": "Implement non-generic counterpart",
            "toolName": "Roslynator"
          }
        ]
      }
    }
}
```

APPENDIX B

C# OPERATIONALIZED MODEL: PRODUCT FACTOR DESCRIPTIONS

This appendix given the descriptions to the product factors described in section 5.2. These definitions match the *"description"* field of the quality model description file described in the operationalized model of chapter 5.

Data Type Integrity A representation of how well each object adheres to C# object oriented typing expectations.

Documentation Information about the source code intended to assist human understanding.

Encryption The concealment of code elements such that external entities are unable to decipher its information.

Exception Handling Goodness of approach in handling exceptions in a compliant and productive way.

Functional Semantics The meaning and intended output of functional acts. For example, dividing all items in a list by two versus multiplying all items by two have different functional semantics.

Functional Syntax The operators, syntactic decision, object selection, etc., that determine how a functional act is defined. This is disjoint from the 'Format' property by having potential impact on more than just human-readability.

Format Code formatting. This involves how the code is written as it appears to a human reader. Common synonyms include 'Style', 'Simplification', 'Readability', 'Convention', 'Verbosity', 'Naming'.

Input/Output Handling Management of the data going into and out of a source component. Can relate to data sanitization, filtering, rejection, querying, database queries and storage, etc.

Resource Handling Management of resources: often related to locking of instances and file I/O.

Structure The organization of classes, files, and methods. Includes object oriented concepts such as inheritance.

APPENDIX C

PIQUE TOOL INTEGRATION TECHNICAL DOCUMENT

This guide walks through the steps necessary to add a third-party C# static analysis tool.

1. Find a desired static analysis tool.
   (a) Browse `https://endler.dev/awesome-static-analysis/` for an appropriate tool.
   (b) Find tool's NuGet page (`https://www.nuget.org/packages/Microsoft.VisualStudio.Threading.Analyzers`).
   (c) Manually get the .dll's for the tool.
      i. `.\nuget.exe install <Package.Name>`.
      ii. Grab .dll files inside new Package.Name directory sibling to nuget.exe location.
   (d) Test: Manually run the tool on cooked project.
      i. Put .dll's in Roslynator's bin directory
      ii. Run Roslynator analysis with following cmd commands: (be sure to cook analysis project to get findings)
         A. SET exe="C:\path\to\Roslynator.exe"
         B. SET assembly="C:\path\to\Roslynator\folder"
         C. SET msbuild="C:\path\to\MSBuild\Current\Bin"
         D. SET output="C:\anything\roslyn.xml"
         E. SET target="C:\path\to\project.sln"
         F. %exe% analyze –analyzer-assemblies=%assembly% –msbuild-path=%msbuild% –output=%output% %target%
2. Add tool as PIQUE project resource.
   (a) Add tool to the PIQUE language specific extension project's resources folder
      i. msusel-pique-csharp/src/main/resources/Roslynator/bin/**
3. Override analyzer interface methods.
   (a) Create class that extends the ITool interface
      i. public class MyAnalyzer implements ITool
   (b) Override the analyze() method; define how to run the tool, return the path to the tool run's output file. See code listing 2.
   (c) Override the parseAnalysis() method; define how to transform the tool's output file findings into Diagnostic objects

   i. See pique-csharp::RoslynatorAnalyzer.parseAnalysis(Path path) for an example. Notable relevant parts are showing in listing 3:

4. Add desired diagnostics (rule findings) to your quality model description under the desired measure(s). For example, in iso25k_csharp_qm_description.qm, see listing 4 and 5 for a before and after.

5. Run model derivation (on benchmark repository) to update model with awareness of the new tool and findings.

```
pb = new ProcessBuilder("cmd.exe", "/c", myAnalyzerExe, commandString,
 ↪  assemblyDir, msBuildPath, outputPath, target)
BufferedReader stdInput = new BufferedReader(new
 ↪  InputStreamReader(p.getInputStream()));
while ((line = stdInput.readLine()) != null) {
    System.out.println("roslynator: " + line);
}
p.waitFor();
return outputPath;
```

Listing 2: Analyze method override.

```
Map<String, Diagnostic> diagnostics = new HashMap<>();
XPathExpression expr = xpath.compile("//Diagnostics/Diagnostic");
String diagnosticId = ((DeferredElementImpl)
 ↪  diagnosticElement).getAttributeNode("Id").getValue();
// attach findings
```

Listing 3: Parse method override.

```
"measure": {
"name": "Resource Handling Findings",
"description": "Description.",
"diagnostics": [
    {
        "name": "RCS1059",
        "description": "Avoid locking on publicly accessible
                        instance",
        "toolName": "Roslynator"
    },
...
```

Listing 4: iso25k_csharp_qm_description, before

```
"measure": {
"name": "Resource Handling Findings",
"description": "Description.",
"diagnostics": [
    {
        "name": "VSTHRD001",
        "description": "Avoid legacy thread switching methods",
        "toolName": "Roslynator"
    },
    ...
    {
        "name": "RCS1059",
        "description": "Avoid locking on publicly accessible
                        instance",
        "toolName": "Roslynator"
    },
...
```

Listing 5: iso25k_csharp_qm_description, after

APPENDIX D

C# BENCHMARK REPOSITORY INFORMATION

The following projects are used as the benchmark repository for the C# operationalization of PIQUE as described in chapter 5. In total, the projects represent ? lines of code and ? logical lines of code.

Table D.1: C# Benchmark Repository Projects

Project	*LoC*	*Empty lines*	*Comment lines*	*Pre-proc. lines*	*Total lines*
ArchiSteamFarm	33,364	11,807	5,335	268	50,774
aspnetboilerplate	108,836	23,775	19,461	327	152,399
AutoMapper	91,428	17,239	3,235	16	111,919
Avalonia	248,486	49,287	38,383	514	336,670
BenchmarkDotNet	64,498	12,390	5,756	291	82,935
CefSharp	19,192	5,118	14,756	30	39,096
choco	37,695	6,713	7,943	77	52,428
CodeHub	433	62	9	0	504
Electron.NET	7,455	1,918	4,033	0	13,406
Entitas-CSharp	14,673	3,923	419	12	19,027
eShopOnContainers	26,109	4,863	742	0	31,714
example-voting-app	652	109	1	0	762
FASTER	41,827	8,678	10,817	467	61,789
FluentTerminal	18,265	3,818	523	148	22,754
FluentValidation	44,343	11,305	13,869	1,068	70,585
graphql-dotnet	41,998	7,003	2,198	7	51,206
Hangfire	148,574	33,780	36,739	1,457	220,550
Humanizer	97,977	12,712	8,740	80	119,509
iotedge	141,812	25,111	9,310	45	176,278
jellyfin	144,467	33,135	28,944	894	207,440
Live-Charts	40,686	10,082	21,809	683	73,260
machinelearning	286,150	53,319	66,858	1,111	407,438
MahApps.Metro	141,065	25,776	26,958	525	194,324
mRemoteNG	48,073	7,513	2,600	630	58,816
msbuild	791,934	153,239	261,471	13,937	1,220,581
NLog	282,992	51,527	215,403	7,411	557,333
Ocelot	44,923	6,614	489	10	52,036
Ombi	36,168	4,910	4,417	122	45,617
OpenRA	127,159	30,137	15,236	2,642	175,174
Opserver	38,507	5,559	3,778	46	47,890
Polly	182,463	51,811	41,344	2,941	278,559
PushSharp	3,448	1,085	450	16	4,999
ql	20,607	4,300	3,750	40	28,697
QuickLook	6,794	1,564	2,152	44	10,554

refit	25,775	6,587	1,639	101	34,102
RestSharp	29,629	8,181	7,782	244	45,836
ScreenToGif	48,303	14,863	10,267	1,936	75,369
server	35,928	4,906	968	4	41,806
ServiceStack	286,151	55,926	37,132	2,597	381,806
ShareX	79,845	17,237	16,587	1,998	115,667
SparkleShare	10,497	4,086	1,272	28	15,883
StackExchange.Redis	115,629	20,398	27,099	739	163,865
workflow-core	20,661	3,749	684	28	25,122
Wox	12,157	1,747	769	132	14,805

Project URLs

ArchiSteamFarm: `https://github.com/JustArchiNET/ArchiSteamFarm`

aspnet-boilerplate: `https://github.com/aspnetboilerplate/aspnetboilerplate`

AutoMapper: `https://github.com/AutoMapper/AutoMapper`

Avalonia: `https://github.com/AvaloniaUI/Avalonia`

BenchmarkDotNet: `https://github.com/dotnet/BenchmarkDotNet`

CefSharp: `https://github.com/cefsharp/CefSharp`

choco: `https://github.com/chocolatey/choco`

CodeHub: `https://github.com/CodeHubApp/CodeHub`

Electron.NET: `https://github.com/ElectronNET/Electron.NET`

Entitas-CSharp: `https://github.com/sschmid/Entitas-CSharp`

eShopOnContainers: `https://github.com/dotnet-architecture/eShopOnContainers`

example-voting-app: `https://github.com/dockersamples/example-voting-app`

FASTER: `https://github.com/microsoft/FASTER`

FluentTerminal: `https://github.com/felixse/FluentTerminal`

FluentValidation: `https://github.com/JeremySkinner/FluentValidation`

graphql-dotnet: `https://github.com/graphql-dotnet/graphql-dotnet`

Hangfire: `https://github.com/HangfireIO/Hangfire`

Humanizer: https://github.com/Humanizr/Humanizer

iotedge: https://github.com/Azure/iotedge

jellyfin: https://github.com/jellyfin/jellyfin

Live-Charts: https://github.com/Live-Charts/Live-Charts

machinelearning: https://github.com/dotnet/machinelearning

MahApps.Metro: https://github.com/MahApps/MahApps.Metro

mRemoteNG: https://github.com/mRemoteNG/mRemoteNG

msbuild: https://github.com/microsoft/msbuild

NLog: https://github.com/NLog/NLog

Ocelot: https://github.com/ThreeMammals/Ocelot

Ombi: https://github.com/tidusjar/Ombi

OpenRA: https://github.com/OpenRA/OpenRA

Opserver: https://github.com/opserver/Opserver

Polly: https://github.com/App-vNext/Polly

PushSharp: https://github.com/Redth/PushSharp

ql: https://github.com/Semmle/ql

QuickLook: https://github.com/QL-Win/QuickLook

refit: https://github.com/reactiveui/refit

RestSharp: https://github.com/restsharp/RestSharp

ScreenToGif: https://github.com/NickeManarin/ScreenToGif

server: https://github.com/bitwarden/server

ServiceStack: https://github.com/ServiceStack/ServiceStack

ShareX: https://github.com/ShareX/ShareX

SparkleShare: https://github.com/hbons/SparkleShare

StackExchange.Redis: https://github.com/StackExchange/StackExchange.Redis

workflow-core: https://github.com/danielgerlag/workflow-core

Wox: https://github.com/Wox-launcher/Wox

APPENDIX E

C# OPERATIONALIZED MODEL: COMPARISON MATRICES

This appendix contains the comparison matrix tables used for factor weight derivation of the operationalized C# discussed in chapter 5. The cell values represent VL: very low. SL: somewhat low. EQ: equal. SH: somewhat high. VH: very high.

E.1 Quality Aspect to TQI layer

Table E.1: TQI Comparison matrix from practitioner interaction

TQI	Comp.	Maint.	Perf.	Port.	Rel.	Sec.	Use.
Comp.	-	SL	SL	SL	VL	VH	EQ
Maint.	-	-	EQ	SH	SL	VH	SH
Perf.	-	-	-	SH	SL	VH	EQ
Port.	-	-	-	-	VL	VH	EQ
Rel.	-	-	-	-	-	VH	SH
Sec.	-	-	-	-	-	-	VL
Use.	-	-	-	-	-	-	-

E.2 Product Factors to Quality Aspects layer

The product factor name abbreviations in the following tables are as follows:

DTI: Data Type Integrity

DOC: Documentation

ENC: Encryption

EXH: Exception Handling

FMT: Format

FNS: Functional Syntax

IOH: Input/Output Handling

RSH: Resource Handling

STR: Structure

Table E.2: Product factors → quality aspect matrix in the context of compatibility

Comp.	DTI	DOC	ENC	EXH	FMT	FNS	IOH	RSH	STR
DTI	-	SH	VH	EQ	VH	EQ	SL	EQ	VH
DOC	-	-	EQ	VL	SH	SL	SL	VL	EQ
ENC	-	-	-	VL	EQ	EQ	VL	VL	EQ
EXH	-	-	-	-	VH	VH	SH	SH	VH
FMT	-	-	-	-	-	EQ	VL	VL	EQ
FNS	-	-	-	-	-	-	SL	SL	EQ
IOH	-	-	-	-	-	-	-	EQ	VH
RSH	-	-	-	-	-	-	-	-	VH
STR	-	-	-	-	-	-	-	-	-

Table E.3: Product factors → quality aspect matrix in the context of maintainability

Maint.	DTI	DOC	ENC	EXH	FMT	FNS	IOH	RSH	STR
DTI	-	EQ	VH	EQ	EQ	EQ	SH	SH	EQ
DOC	-	-	VH	SH	EQ	EQ	SH	VH	EQ
ENC	-	-	-	VL	VL	VL	VL	VL	VL
EXH	-	-	-	-	SL	SL	SH	SH	SL
FMT	-	-	-	-	-	EQ	SH	VH	EQ
FNS	-	-	-	-	-	-	SH	VH	EQ
IOH	-	-	-	-	-	-	-	EQ	SL
RSH	-	-	-	-	-	-	-	-	VL
STR	-	-	-	-	-	-	-	-	-

Table E.4: Product factors → quality aspect matrix in the context of performance efficiency

Perf.	DTI	DOC	ENC	EXH	FMT	FNS	IOH	RSH	STR
DTI	-	VH	SH	SL	VH	VH	SL	SL	SH
DOC	-	-	SL	VL	EQ	EQ	VL	VL	VL
ENC	-	-	-	VL	EQ	EQ	VL	VL	SL
EXH	-	-	-	-	VH	SH	EQ	EQ	VH
FMT	-	-	-	-	-	EQ	VL	VL	SL
FNS	-	-	-	-	-	-	SL	SL	SL
IOH	-	-	-	-	-	-	-	EQ	VH
RSH	-	-	-	-	-	-	-	-	VH
STR	-	-	-	-	-	-	-	-	-

Table E.5: Product factors → quality aspect matrix in the context of portability

Port.	DTI	DOC	ENC	EXH	FMT	FNS	IOH	RSH	STR
DTI	-	SL	SH	SL	EQ	EQ	VL	SL	EQ
DOC	-	-	SH	SL	SH	EQ	SL	SL	SH
ENC	-	-	-	VL	EQ	SL	VL	SL	EQ
EXH	-	-	-	-	SH	SH	EQ	EQ	SH
FMT	-	-	-	-	-	SL	VL	VL	EQ
FNS	-	-	-	-	-	-	VL	VL	EQ
IOH	-	-	-	-	-	-	-	EQ	SH
RSH	-	-	-	-	-	-	-	-	SH
STR	-	-	-	-	-	-	-	-	-

Table E.6: Product factors → quality aspect matrix in the context of reliability

Rel.	DTI	DOC	ENC	EXH	FMT	FNS	IOH	RSH	STR
DTI	-	VH	VH	SL	VH	SH	SL	SL	VH
DOC	-	-	VH	VL	VH	EQ	VL	VL	VH
ENC	-	-	-	VL	EQ	SL	VL	VL	SL
EXH	-	-	-	-	VH	SH	EQ	EQ	VH
FMT	-	-	-	-	-	SL	VL	VL	SL
FNS	-	-	-	-	-	-	SL	SL	SL
IOH	-	-	-	-	-	-	-	EQ	VL
RSH	-	-	-	-	-	-	-	-	VL
STR	-	-	-	-	-	-	-	-	-

Table E.7: Product factors → quality aspect matrix in the context of security

Sec.	DTI	DOC	ENC	EXH	FMT	FNS	IOH	RSH	STR
DTI	-	VH	SL	EQ	VH	VH	SL	EQ	VH
DOC	-	-	VL	VL	EQ	EQ	VL	VL	EQ
ENC	-	-	-	SH	VH	VH	EQ	SH	VH
EXH	-	-	-	-	VH	VH	SL	EQ	VH
FMT	-	-	-	-	-	EQ	VL	VL	EQ
FNS	-	-	-	-	-	-	VL	VL	EQ
IOH	-	-	-	-	-	-	-	SH	VH
RSH	-	-	-	-	-	-	-	-	VH
STR	-	-	-	-	-	-	-	-	-

Table E.8: Product factors → quality aspect matrix in the context of usability

Use.	DTI	DOC	ENC	EXH	FMT	FNS	IOH	RSH	STR
DTI	-	SL	EQ	VL	EQ	EQ	VL	EQ	EQ
DOC	-	-	SH	VL	SH	SH	VL	SH	SH
ENC	-	-	-	VL	EQ	EQ	VL	EQ	EQ
EXH	-	-	-	-	VH	VH	EQ	VH	VH
FMT	-	-	-	-	-	EQ	VH	EQ	EQ
FNS	-	-	-	-	-	-	VL	EQ	EQ
IOH	-	-	-	-	-	-	-	VH	VH
RSH	-	-	-	-	-	-	-	-	EQ
STR	-	-	-	-	-	-	-	-	-

APPENDIX F

C# OPERATIONALIZED MODEL: DERIVED WEIGHTS

F.1 Derived Factor Incoming Weight Values

TQI Node {Performance Efficiency: 0.151, Portability: 0.0816, Maintainability: 0.1822, Compatibility: 0.0537, Reliability: 0.4327, Security: 0.0156, Usability: 0.0831}

Performance Efficiency {Functional Syntax: 0.0296, Format: 0.0198, Resource Handling: 0.2408, Data Type Integrity: 0.1229, Documentation: 0.0170, I/O Handling: 0.2408, Exception Handling: 0.2408, Encryption: 0.0265, Structure: 0.0617}

Portability {Functional Syntax: 0.0575, Format: 0.0292, Resource Handling: 0.2351, Data Type Integrity: 0.0449, Documentation: 0.0909, I/O Handling: 0.2714, Exception Handling: 0.2042, Encryption: 0.0262, Structure: 0.0407}

Maintainability {Functional Syntax: 0.1747, Format: 0.1747, Resource Handling: 0.0321, Data Type Integrity: 0.1339, Documentation: 0.1747, I/O Handling: 0.0419, Exception Handling: 0.0808, Encryption: 0.0124, Structure: 0.1747}

Compatibility {Functional Syntax: 0.0518, Format: 0.0221, Resource Handling: 0.1701, Data Type Integrity: 0.1488, Documentation: 0.0318, I/O Handling: 0.1983, Exception Handling: 0.3283, Encryption: 0.0244, Structure: 0.0244}

Reliability {Functional Syntax: 0.0265, Format: 0.0115, Resource Handling: 0.1318, Data Type Integrity: 0.1924, Documentation: 0.0275, I/O Handling: 0.1318, Exception Handling: 0.2606, Encryption: 0.0115, Structure: 0.2063}

Security {Functional Syntax: 0.0184, Format: 0.0184, Resource Handling: 0.1219, Data Type Integrity: 0.1219, Documentation: 0.0184, I/O Handling: 0.2803, Exception Handling: 0.1219, Encryption: 0.2803, Structure: 0.0184}

Usability {Functional Syntax: 0.0312, Format: 0.1762, Resource Handling: 0.0312, Data Type Integrity: 0.0312, Documentation: 0.1111, I/O Handling: 0.2205, Exception Handling: 0.3363, Encryption: 0.0312, Structure: 0.0312}

APPENDIX G

C# OPERATIONALIZED MODEL: FULL MODEL DATA

```
{
  "name": "CSharp ISO25k Quality Model",
  "additionalData": {},
  "global_config": {
    "benchmark_strategy": "pique.calibration.DefaultBenchmarker",
    "weights_strategy": "pique.calibration.AHPWeighter"
  },
  "factors": {
    "product_factors": {
      "Functional Syntax": {
        "description": "The operators, syntactic decision, object
        ↪  selection, etc., that determine how a functional act is
        ↪  defined. This is disjoint from the Format property by having
        ↪  potential impact on more than just human-readability",
        "value": 0.0
      },
      "Format": {
        "description": "Code formatting. This involves how the code is
        ↪  written as it appears to a human reader. Common synonyms
        ↪  include Style, Simplification, Readability, Convention,
        ↪  Verbosity, Naming",
        "value": 0.0
      },
      "Resource Handling": {
        "description": "Management of resources: often related to locking
        ↪  of instances and file IO",
        "value": 0.0
      },
      "Data Type Integrity": {
        "description": "A  representation of how well each object adheres
        ↪  to CSharp object oriented typing expectations",
        "value": 0.0
      },
      "Documentation": {
        "description": "Information about the source code intended to
        ↪  assist human understanding",
        "value": 0.0
      },
      "IO Handling": {
        "description": "Management of the data going into and out of a
        ↪  source component.  Can relate to data sanitization, filtering,
        ↪  rejection, querying, database queries and storage, etc",
        "value": 0.0
      },
      "Exception Handling": {
```

```
        "description": "Goodness of approach in handling exceptions in a compliant and productive way",
        "value": 0.0
      },
      "Encryption": {
        "description": "The concealment of code elements such that external entities are unable to decipher its information",
        "value": 0.0
      },
      "Structure": {
        "description": "The organization of classes, files, and methods. Includes object oriented concepts such as inheritance",
        "value": 0.0
      }
    },
    "quality_aspects": {
      "Performance Efficiency": {
        "weights": {
          "Functional Syntax": 0.0296,
          "Format": 0.0198,
          "Resource Handling": 0.2408,
          "Data Type Integrity": 0.1229,
          "Documentation": 0.017,
          "IO Handling": 0.2408,
          "Exception Handling": 0.2408,
          "Encryption": 0.0265,
          "Structure": 0.0617
        },
        "description": "This characteristic represents the performance relative to the amount of resources used under stated conditions",
        "value": 0.0
      },
      "Portability": {
        "weights": {
          "Functional Syntax": 0.0575,
          "Format": 0.0292,
          "Resource Handling": 0.2351,
          "Data Type Integrity": 0.0449,
          "Documentation": 0.0909,
          "IO Handling": 0.2714,
          "Exception Handling": 0.2042,
          "Encryption": 0.0262,
          "Structure": 0.0407
        },
```

```
      "description": "Degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another",
      "value": 0.0
    },
    "Maintainability": {
      "weights": {
        "Functional Syntax": 0.1747,
        "Format": 0.1747,
        "Resource Handling": 0.0321,
        "Data Type Integrity": 0.1339,
        "Documentation": 0.1747,
        "IO Handling": 0.0419,
        "Exception Handling": 0.0808,
        "Encryption": 0.0124,
        "Structure": 0.1747
      },
      "description": "This characteristic represents the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements",
      "value": 0.0
    },
    "Compatibility": {
      "weights": {
        "Functional Syntax": 0.0518,
        "Format": 0.0221,
        "Resource Handling": 0.1701,
        "Data Type Integrity": 0.1488,
        "Documentation": 0.0318,
        "IO Handling": 0.1983,
        "Exception Handling": 0.3283,
        "Encryption": 0.0244,
        "Structure": 0.0244
      },
      "description": "Degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions while sharing the same hardware or software environment",
      "value": 0.0
    },
    "Reliability": {
      "weights": {
        "Functional Syntax": 0.0265,
```

```
            "Format": 0.0115,
            "Resource Handling": 0.1318,
            "Data Type Integrity": 0.1924,
            "Documentation": 0.0275,
            "IO Handling": 0.1318,
            "Exception Handling": 0.2606,
            "Encryption": 0.0115,
            "Structure": 0.2063
          },
          "description": "Degree to which a system, product or component performs specified functions under specified conditions for a specified period of time",
          "value": 0.0
        },
        "Security": {
          "weights": {
            "Functional Syntax": 0.0184,
            "Format": 0.0184,
            "Resource Handling": 0.1219,
            "Data Type Integrity": 0.1219,
            "Documentation": 0.0184,
            "IO Handling": 0.2803,
            "Exception Handling": 0.1219,
            "Encryption": 0.2803,
            "Structure": 0.0184
          },
          "description": "Degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization",
          "value": 0.0
        },
        "Usability": {
          "weights": {
            "Functional Syntax": 0.0312,
            "Format": 0.1762,
            "Resource Handling": 0.0312,
            "Data Type Integrity": 0.0312,
            "Documentation": 0.1111,
            "IO Handling": 0.2205,
            "Exception Handling": 0.3363,
            "Encryption": 0.0312,
            "Structure": 0.0312
          },
```

```
        "description": "Degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use",
        "value": 0.0
      }
    },
    "tqi": {
      "Total Software Quality": {
        "weights": {
          "Performance Efficiency": 0.151,
          "Portability": 0.0816,
          "Maintainability": 0.1822,
          "Compatibility": 0.0537,
          "Reliability": 0.4327,
          "Security": 0.0156,
          "Usability": 0.0831
        },
        "description": "The total quality value representing the entire system",
        "value": 0.0
      }
    }
  },
  "measures": {
    "Exception Handling Smells": {
      "positive": false,
      "normalizer": "pique.evaluation.DefaultNormalizer",
      "eval_strategy": "pique.evaluation.DefaultMeasureEvaluator",
      "parents": [
        "Exception Handling"
      ],
      "diagnostics": [
        {
          "name": "loc",
          "eval_strategy": "pique.evaluation.LOCDiagnosticEvaluator",
          "toolName": "RoslynatorLoc",
          "findings": [],
          "description": "Normalizer diagnostic lines of code",
          "value": 0.0
        },
        {
          "name": "RCS1044",
          "eval_strategy": "pique.evaluation.DefaultDiagnosticEvaluator",
          "toolName": "Roslynator",
          "findings": [],
```

```
            "description": "Remove original exception from throw statement",
            "value": 0.0
          },
          {
            "name": "RCS1075",
            "eval_strategy": "pique.evaluation.DefaultDiagnosticEvaluator",
            "toolName": "Roslynator",
            "findings": [],
            "description": "Avoid empty catch clause that catches
            ↪  System.Exception",
            "value": 0.0
          },
          // ...
          {
            "name": "RCS1236",
            "eval_strategy": "pique.evaluation.DefaultDiagnosticEvaluator",
            "toolName": "Roslynator",
            "findings": [],
            "description": "Use exception filter",
            "value": 0.0
          }
        ],
        "thresholds": [
          0.0,
          9.476E-5,
          6.38E-4
        ],
        "description": "Code smells related to exception handling",
        "value": 0.0
      },
      "Functional Syntax Smells": {
        "positive": false,
        "normalizer": "pique.evaluation.DefaultNormalizer",
        "eval_strategy": "pique.evaluation.DefaultMeasureEvaluator",
        "parents": [
          "Functional Syntax"
        ],
        "diagnostics": [
          {
            "name": "loc",
            "eval_strategy": "pique.evaluation.LOCDiagnosticEvaluator",
            "toolName": "RoslynatorLoc",
            "findings": [],
            "description": "Normalizer diagnostic lines of code",
            "value": 0.0
```

```
        },
        {
          "name": "RCS1048",
          "eval_strategy": "pique.evaluation.DefaultDiagnosticEvaluator",
          "toolName": "Roslynator",
          "findings": [],
          "description": "Use lambda expression instead of anonymous
          ↪  method",
          "value": 0.0
        },
        {
          "name": "RCS1061",
          "eval_strategy": "pique.evaluation.DefaultDiagnosticEvaluator",
          "toolName": "Roslynator",
          "findings": [],
          "description": "Merge if statement with nested if statement",
          "value": 0.0
        },
        // ...
        {
          "name": "RCS1236",
          "eval_strategy": "pique.evaluation.DefaultDiagnosticEvaluator",
          "toolName": "Roslynator",
          "findings": [],
          "description": "Use exception filter",
          "value": 0.0
        }
      ],
      "thresholds": [
        0.0,
        0.00140055,
        0.00696056
      ],
      "description": "Code smells related to functional syntax",
      "value": 0.0
    },
    "Encryption Smells": {
      "positive": false,
      "normalizer": "pique.evaluation.DefaultNormalizer",
      "eval_strategy": "pique.evaluation.DefaultMeasureEvaluator",
      "parents": [
        "Encryption"
      ],
      "diagnostics": [
        {
```

```
            "name": "loc",
            "eval_strategy": "pique.evaluation.LOCDiagnosticEvaluator",
            "toolName": "RoslynatorLoc",
            "findings": [],
            "description": "Normalizer diagnostic lines of code",
            "value": 0.0
          },
          // ...
          {
            "name": "SCS0034",
            "eval_strategy": "pique.evaluation.DefaultDiagnosticEvaluator",
            "toolName": "Roslynator",
            "findings": [],
            "description": "Password RequiredLength Not Set",
            "value": 0.0
          }
        ],
        "thresholds": [
          0.0,
          0.0,
          0.0
        ],
        "description": "Code smells related to Encryption",
        "value": 0.0
      },
      "Documentation Smells": {
        "positive": false,
        "normalizer": "pique.evaluation.DefaultNormalizer",
        "eval_strategy": "pique.evaluation.DefaultMeasureEvaluator",
        "parents": [
          "Documentation"
        ],
        "diagnostics": [
          {
            "name": "loc",
            "eval_strategy": "pique.evaluation.LOCDiagnosticEvaluator",
            "toolName": "RoslynatorLoc",
            "findings": [],
            "description": "Normalizer diagnostic lines of code",
            "value": 0.0
          },
          {
            "name": "RCS1138",
            "eval_strategy": "pique.evaluation.DefaultDiagnosticEvaluator",
            "toolName": "Roslynator",
```

```
          "findings": [],
          "description": "Add summary to documentation comment",
          "value": 0.0
        },
        {
          "name": "RCS1139",
          "eval_strategy": "pique.evaluation.DefaultDiagnosticEvaluator",
          "toolName": "Roslynator",
          "findings": [],
          "description": "Add summary element to documentation comment",
          "value": 0.0
        },
        // ...
        {
          "name": "RCS1232",
          "eval_strategy": "pique.evaluation.DefaultDiagnosticEvaluator",
          "toolName": "Roslynator",
          "findings": [],
          "description": "Order elements in documentation comment",
          "value": 0.0
        }
      ],
      "thresholds": [
        0.0,
        3.2634E-4,
        0.00205573
      ],
      "description": "Code smells related to documentation",
      "value": 0.0
    },
    "Format Smells": {
      "positive": false,
      "normalizer": "pique.evaluation.DefaultNormalizer",
      "eval_strategy": "pique.evaluation.DefaultMeasureEvaluator",
      "parents": [
        "Format"
      ],
      "diagnostics": [
        {
          "name": "loc",
          "eval_strategy": "pique.evaluation.LOCDiagnosticEvaluator",
          "toolName": "RoslynatorLoc",
          "findings": [],
          "description": "Normalizer diagnostic lines of code",
          "value": 0.0
```

```
        },
        {
          "name": "RCS1001",
          "eval_strategy": "pique.evaluation.DefaultDiagnosticEvaluator",
          "toolName": "Roslynator",
          "findings": [],
          "description": "Add braces (when expression spans over multiple
          ↪  lines)",
          "value": 0.0
        },
        {
          "name": "RCS1002",
          "eval_strategy": "pique.evaluation.DefaultDiagnosticEvaluator",
          "toolName": "Roslynator",
          "findings": [],
          "description": "Remove braces",
          "value": 0.0
        },
        // ...
        {
          "name": "RCS1215",
          "eval_strategy": "pique.evaluation.DefaultDiagnosticEvaluator",
          "toolName": "Roslynator",
          "findings": [],
          "description": "Expression is always equal to true/false",
          "value": 0.0
        }
      ],
      "thresholds": [
        0.00107901,
        0.04671379,
        0.12169017
      ],
      "description": "Code smells related to format",
      "value": 0.0
    },
    "DTI Smells": {
      "positive": false,
      "normalizer": "pique.evaluation.DefaultNormalizer",
      "eval_strategy": "pique.evaluation.DefaultMeasureEvaluator",
      "parents": [
        "Data Type Integrity"
      ],
      "diagnostics": [
        {
```

```
            "name": "loc",
            "eval_strategy": "pique.evaluation.LOCDiagnosticEvaluator",
            "toolName": "RoslynatorLoc",
            "findings": [],
            "description": "Normalizer diagnostic lines of code",
            "value": 0.0
          },
          {
            "name": "RCS1008",
            "eval_strategy": "pique.evaluation.DefaultDiagnosticEvaluator",
            "toolName": "Roslynator",
            "findings": [],
            "description": "Use explicit type instead of 'var' (when the type
            ↪  is not obvious)",
            "value": 0.0
          },
          {
            "name": "RCS1009",
            "eval_strategy": "pique.evaluation.DefaultDiagnosticEvaluator",
            "toolName": "Roslynator",
            "findings": [],
            "description": "Use explicit type instead of 'var' (foreach
            ↪  variable)",
            "value": 0.0
          },
          // ...
          {
            "name": "RCS1234",
            "eval_strategy": "pique.evaluation.DefaultDiagnosticEvaluator",
            "toolName": "Roslynator",
            "findings": [],
            "description": "Duplicate enum value",
            "value": 0.0
          }
        ],
        "thresholds": [
          0.0,
          0.0,
          1.398E-5
        ],
        "description": "Code smells related to data type integrity",
        "value": 0.0
      },
      "Resource Handling Smells": {
        "positive": false,
```

```
      "normalizer": "pique.evaluation.DefaultNormalizer",
      "eval_strategy": "pique.evaluation.DefaultMeasureEvaluator",
      "parents": [
        "Resource Handling"
      ],
      "diagnostics": [
        {
          "name": "loc",
          "eval_strategy": "pique.evaluation.LOCDiagnosticEvaluator",
          "toolName": "RoslynatorLoc",
          "findings": [],
          "description": "Normalizer diagnostic lines of code",
          "value": 0.0
        },
        {
          "name": "RCS1059",
          "eval_strategy": "pique.evaluation.DefaultDiagnosticEvaluator",
          "toolName": "Roslynator",
          "findings": [],
          "description": "Avoid locking on publicly accessible instance",
          "value": 0.0
        },
        {
          "name": "RCS1090",
          "eval_strategy": "pique.evaluation.DefaultDiagnosticEvaluator",
          "toolName": "Roslynator",
          "findings": [],
          "description": "Call 'ConfigureAwait(false)'",
          "value": 0.0
        },
        // ...
        {
          "name": "VSTHRD200",
          "eval_strategy": "pique.evaluation.DefaultDiagnosticEvaluator",
          "toolName": "Roslynator",
          "findings": [],
          "description": "Use Async naming convention",
          "value": 0.0
        }
      ],
      "thresholds": [
        0.0,
        0.00409824,
        0.03116094
      ],
```

```
502         "description": "Code smells related to resource handling",
503         "value": 0.0
504       },
505       "IO Handling Smells": {
506         "positive": false,
507         "normalizer": "pique.evaluation.DefaultNormalizer",
508         "eval_strategy": "pique.evaluation.DefaultMeasureEvaluator",
509         "parents": [
510           "IO Handling"
511         ],
512         "diagnostics": [
513           {
514             "name": "loc",
515             "eval_strategy": "pique.evaluation.LOCDiagnosticEvaluator",
516             "toolName": "RoslynatorLoc",
517             "findings": [],
518             "description": "Normalizer diagnostic lines of code",
519             "value": 0.0
520           },
521           {
522             "name": "SCS0001",
523             "eval_strategy": "pique.evaluation.DefaultDiagnosticEvaluator",
524             "toolName": "Roslynator",
525             "findings": [],
526             "description": "Command Injection",
527             "value": 0.0
528           },
529           {
530             "name": "SCS0002",
531             "eval_strategy": "pique.evaluation.DefaultDiagnosticEvaluator",
532             "toolName": "Roslynator",
533             "findings": [],
534             "description": "SQL Injection (LINQ)",
535             "value": 0.0
536           },
537           // ...
538           {
539             "name": "SCS0036",
540             "eval_strategy": "pique.evaluation.DefaultDiagnosticEvaluator",
541             "toolName": "Roslynator",
542             "findings": [],
543             "description": "SQL Injection (EnterpriseLibrary.Data)",
544             "value": 0.0
545           }
546         ],
```

```
      "thresholds": [
        0.0,
        0.0,
        0.0
      ],
      "description": "Code smells related to IO handling",
      "value": 0.0
    },
    "Structure Smells": {
      "positive": false,
      "normalizer": "pique.evaluation.DefaultNormalizer",
      "eval_strategy": "pique.evaluation.DefaultMeasureEvaluator",
      "parents": [
        "Structure"
      ],
      "diagnostics": [
        {
          "name": "loc",
          "eval_strategy": "pique.evaluation.LOCDiagnosticEvaluator",
          "toolName": "RoslynatorLoc",
          "findings": [],
          "description": "Normalizer diagnostic lines of code",
          "value": 0.0
        },
        {
          "name": "RCS1060",
          "eval_strategy": "pique.evaluation.DefaultDiagnosticEvaluator",
          "toolName": "Roslynator",
          "findings": [],
          "description": "Declare each type in separate file",
          "value": 0.0
        },
        {
          "name": "RCS1085",
          "eval_strategy": "pique.evaluation.DefaultDiagnosticEvaluator",
          "toolName": "Roslynator",
          "findings": [],
          "description": "Use auto-implemented property",
          "value": 0.0
        },
        // ...
        {
          "name": "RCS1241",
          "eval_strategy": "pique.evaluation.DefaultDiagnosticEvaluator",
          "toolName": "Roslynator",
```

```
            "findings": [],
            "description": "Implement non-generic counterpart",
            "value": 0.0
          }
        ],
        "thresholds": [
          0.0,
          0.00334999,
          0.00923788
        ],
        "description": "Code smells related to structure",
        "value": 0.0
      }
    }
  }
```

APPENDIX H

THE QUAMOCO META MODEL UML

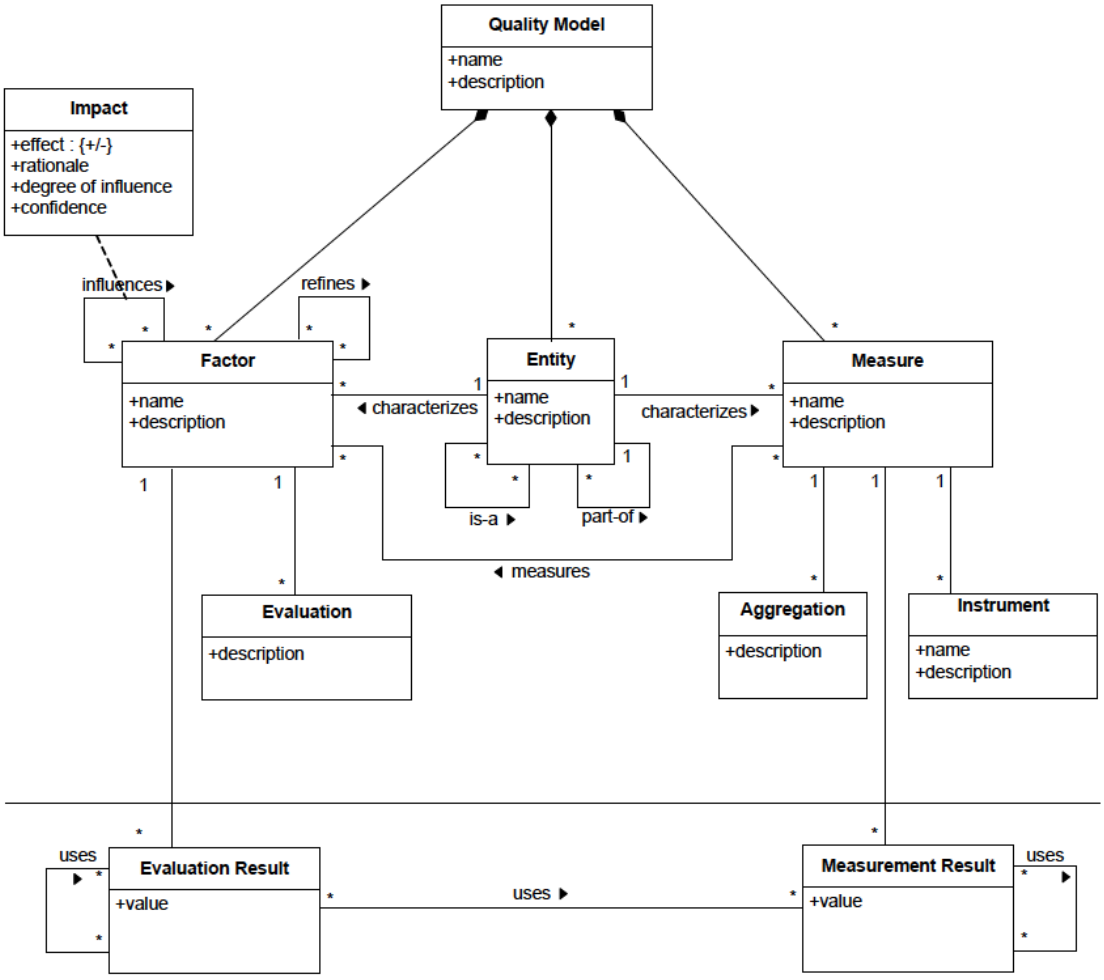

Figure H.1: Quamoco meta model. Source: [41]