Université de Montréal

A Quality Model considering Program Architecture

par
Khashayar Khosravi

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès science (M.Sc.)
en informatique

Août, 2005

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé :

A Quality Model considering Program Architecture

présenté par :

Khashayar Khosravi

a été évalué par le jury composé de :

Esma Aïmeur
président–rapporteur

Yann-Gaël Guéhéneuc
directeur de recherche

Houari Sahraoui
membre du jury

Mémoire accepté le

To my Mom and Dad
And also
To my Sister and my Brother
With all my love.

And may we be among those
who make this life fresh!
You, lords of wisdom,
who bring happiness through righteousness,
come, let us be
single-minded in the realm of inner intellect.

(Zoroaster-Gatha: Song 3 - stanza 9)

Hear the best with your ears
and
ponder with a bright mind
Then each man and woman
for his or her self
select either of the [following] two.

(Zoroaster-Gatha: Song 3 - stanza 2)

ACKNOWLEDGEMENTS

May the best of blessings come to the person
who gives blessings to others.
(Zoroaster-Gatha: Song 8.2)

During this hard time, I could not forget to express my gratitude to many people for their friendship, help, and support.

It was a privilege to work with Yann-Gaël Guéhéneuc; thank you for welcoming me in your lab and for introducing me to your research. You are an amazing scientist and an even more amazing person. Thank you for all your help, support, encouragement and great supervision, I always could count on you, especially during my parents unfortunate happenstance, I could not have asked for a better advisor or for a dearer friends. Thank you for everything from the bottom of my heart.

I thank my thesis committee members: Esma Aïmeur, Yann-Gaël Guéhéneuc, and Houari Sahraoui for their help, advice, and patience.

Special thanks to my family, which I value most in my life; you are my source of strength, love, happiness, and support. My Mom Morvarid Hooshidary: You were always there for me during the good times and the bad times; my Dad Esfandyar Khosravi: You provided the foundation and appreciation of hard working, I hope you will recover soon. My lovely sister Artemis Khosravi, my cute little nephew Atria Shenassa, and my brother Bahram Khosrav: You left your job to take care of our parents in their happenstance, you stay with them to make this opportunity for me to finish my thesis, words cannot express my gratitude, I can never thank you enough for your help, encouragement, and support. My brother-in-law Farshid Shenassa and my sister-in-law Faranak Salamati: Thank you for listening, caring, and making me believe that nothing is impossible, I dedicate this dissertation to all of you.

I think that if I honestly reflects on who I am and how I got here, I would like to thank my cousins: Feraidoon Khosravi and Mahbanoo Behboodi, also their family Mehrnaz Khosravi, Farzad Behboodi, Darya and Donya Khosravi, Nairika, Armita and Vesta Behboodi, I started my new life in Canada with you, thanks for your advice, encouragements, and friendship from the very beginning to the very end. My uncle and my aunt Cyrus Khosravi and Shahnaz Kouhenoori: All of you are very dear to my heart.

The work of some unknown person makes our lives easier everyday. I believe it is appropriate to acknowledge all of these unknown persons; but it is also necessary to acknowledge those people that we know have directly shaped our lives and our work at University of Montreal, Department of informatics and operations research, Software engineering group: my dear classmates in GEODES lab., Ptidej Group, Support team,

Secretariat, Library's personnel and all those who participated directly or indirectly even in silence for my personal success, and to those who were the reason to came to university every single day.

Khashayar Khosravi
August 2005

ABSTRACT

Maintenance cost of object-oriented programs during the past decades increased to more than 70% of the overall cost of programs [72, 76]. Maintenance is expensive and tedious because of the difficulty to predict maintenance effort. Quality is the more important part of software development, because high quality in software development could reduce the software development cost dramatically.

Software quality models link internal attributes of programs with external quality characteristics. They help in understanding relationships among internal attributes and between internal attributes and quality characteristics. Object-oriented software quality models usually use metrics on classes (such as number of methods) or on relationships between classes (for example coupling) to measure internal attributes of programs. However, the quality of object-oriented programs does not depend on classes solely: It depends on the organisation of classes—the program architectures—also.

We propose an approach to build quality models using design patterns to consider program architectures. Design patterns are simple and elegant solution to reusing object oriented software designs. We study the gain and loss in software quality when using design patterns, which claim to bring flexibility, elegancy and reusability to software design [38]. First, we evaluate the concrete quality characteristics of design patterns by evaluating programs implemented with and without design patterns manually and using metrics. Then, we study the flexibility, elegancy, and reusability of programs, to assess their coverage of software quality, other quality characteristics and the interrelationships among quality characteristics. We detail the building of a software quality model using design patterns. Finally, we introduce a first case study in building and in applying a quality model using design patterns on the JHotDraw, JUnit, and Lexi programs. This dissertation intends to assist object-oriented software quality measurement, improving software evaluation by building a quality model using design patterns and considering program architectures.

RÉSUMÉ

Le coût de la maintenance des programmes orientés-objets a augmenté ces dernières années de pres de 70% du coût total d'un programme [72, 76]. La maintenance est coûteuse et difficile car la qualité des programmes est souvent faible. La qualité est le facteur le plus important dans le coût de la maintenance car des programmes de très bonnes qualité sont plus faciles à maintenir.

Nous proposons une approche pour construire des modèles de qualité qui utilisent les patrons de conception pour prendre en comptent l'architecture des programmes. Les patrons de conception sont des solutions simples et élégantes à des problèmes récurrents de conception. Aucun modèle de qualité à ce jour ne prend en compte la structure des programmes comme nous le faisons.

D'abord, nous évaluons concrètement la qualité des solutions des patrons de conception. Ensuite, nous étudions les qualités de programmes implantés avec des patrons de conception. Nous détaillons ensuite la construction d'un modèle de qualité utilisant ces études des patrons de conception. Enfin, nous introduisons un cas d'utilisation du modelée de qualité pour évaluer la qualité des programmes JHotDraw, JUnit et Lexi. Nous montrons que notre modèle de qualité aide dans l'évaluation de la qualité des programmes.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

JHotDraw	Java GUI framework for technical and structured graphics.
JUnit	Java testing framework for eXtreme Programming.
Lexi	Java open-source word processor.
Weka	Collection of machine learning algorithms for data mining tasks.
JRip	Class of implementation of a propositional rule learner.
Ripper	Repeated Incremental Pruning to Produce Error Reduction, JRip algorithm.
PADL	Pattern and Abstract-level Description Language, meta-model to describe programs.
POM	Primitives, Operators, Metrics, PADL-based metric framework.
Ptidej	Pattern Trace Identification, Detection, and Enhancement in Java, reverse-engineering and quality evaluation framework.
$\mathcal{BP}$	Simple programs to be considered as sample evaluation set.
$\mathcal{EG}$	Evaluation group, at least one person evaluating $\mathcal{BP}$.
$\mathcal{EP}$	Program that should be evaluated.

CHAPTER 1

INTRODUCTION

Software measurement has become critical issue for good software engineering but the majority published work on software engineering focus on measuring the coding activities. As quality indicators and predictors of structural problems, metrics measurement should be available as early as possible solution, but metrics value are dependant on source code availability.

1.1 Motivation

Software metrics and quality models play a pivotal role in the evaluation of software quality. A number of quality models and software metrics are used to build quality software both in industry and in academia.

However, during our research on evaluating software quality using design patterns, we faced many issues related to existing software metrics and quality models.

1.2 Problem

Software quality models link internal attributes of programs with external quality characteristics. They help in understanding the relationships among internal attributes and between internal attributes and quality characteristics.

Object-oriented software quality models usually use metrics on classes (such as number of methods) or on relationships between classes (*i.e.,* coupling) to measure internal attributes of programs. However, the quality of object-oriented programs does not depend on classes solely: It depends on the organisation of classes—the programs architectures—also. Indeed, architectures are an important quality factor. A "good" architecture makes a program easier to understand and to maintain. In particular, a "good" architecture is characterised by the use of recognised patterns, such as design patterns [38].

Our thesis is that a quality model taking in account structural patterns reflect the quality characteristics of a program better.

1.3 Contributions

We present an approach to build quality models using patterns to consider program architectures. We show that no existing work attempts to build a quality model consider-

ing program architectures and put the based on software design patterns. We justify the use of design patterns to build quality models, describe the advantages and limitations of such an approach, and introduce a first case study in building and in applying a quality model using design patterns on the JHotDraw, JUnit, and Lexi programs. We conclude on the advantages of using design patterns to build software quality models and on the difficulty of doing so. This thesis intends to assist quality evaluation of object-oriented programs by improving method and discusses some issues to present our approach to software quality evaluation.

Our contribution is a thorough study of the state-of-the-art of the quality characteristics of programs, of software metrics, and of design patterns. We also introduce a detailed analysis of design patterns and software metrics. We propose a quality model using design pattern to consider program architecture. To the best of our knowledge, we propose the first quality model built specifically to assess quality characteristics using patterns.

1.4 Organisation

Chapter 2 details the state-of-the-art on quality models and show that no quality models consider the program architectures. Chapter 3 introduces our approach to assessing the quality of programs while considering their architectures. Chapter 4 presents our quality model and details its characteristics and sub-characteristics. Chapter 5 defines the metrics associated with the characteristics and sub-characteristics of our quality models. Chapter 6 provides a thorough study of the quality of design patterns using our quality model according. Chapter 7 uses the results form previous chapters to describe our complete approach and the resulting quality model considering program architectures. Chapter 8 presents an application of our quality model on three programs: JHotDraw, JUnit, and Lexi. Finally, Chapter 9 concludes and introduces future work.

CHAPTER 2

STATE OF THE ART

As software becomes more and more pervasive, there has been a growing concern in the academic community and in the public about software quality. This concern arises from the acknowledgment that the main objective of software industries in non-military sectors is to balance price and quality to stay ahead of competitors.

In past 15 years, the software industry has created many new different markets, such as open source software and commerce over the Internet. With these new markets, customers of programs now have very high expectations on quality and use quality as a major drive in choosing programs.

Some organizations, such as *ISO* and *IEEE*, try to standardise software quality by defining models combining and relating software quality characteristics and sub-characteristics. Meanwhile, researchers propose software metrics as tools to measure programs source code, architecture, and performances. However, there is a not yet clear and consensual relation among software quality models and between models and metrics. Moreover the process of software quality evaluation remains an open issue with many models.

2.1 Brief History

"Walter Shewhart [80] was a statistician at AT&T Bell Laboratories in the 1920s and is regarded as the funder of statistical quality improvement, and modern process improvement is based on the concept of process control developed by Shewhart" [67].

Since 1920, progress in the field of software quality has been time-consuming and ambiguous. Software metrics[1] and quality models[2] are known as major reference in the field of software quality evaluation but there is still no clear methods for assessing software quality via software metrics.

2.2 Why Do We Need Quality?

Everyone agrees that software quality is the most important element in software development because high quality could reduce the cost of maintenance, test, security, and software reusing. But quality has very different meanings for customers (price...),

[1] In appendix I, we introduce a list of metrics from different sources.
[2] In appendix IV, we present quality models which from different sources.

users (usability, understandablity...), management (costs–benefits...), marketing (purposes, functionalities, advertising...), developers (understandability, maintainability...), testers (testability, performances...), and support personnel (portability...). Many institutes and organizations have their own definitions of quality and their own quality characteristics (ISO, IEEE...).

2.3 Quality Affections

The software industry is growing up daily and "it is rather surprising that more serious and definitive work has not been done to date in the area of evaluating software quality" [12]. The reason is that "quality is hard to define, impossible to measure, easy to recognize" [56, 74] and "transparent when presented, but easily recognized in its absence" [39, 74] but "[q]uality is not a single idea, but rather a multidimensional concept. The dimensions of quality include the entity of interest, the viewpoint on that entity, and quality attributes of that entity" [51].

2.4 Quality Definitions

Some organisations try to develop standard definitions for quality. We present some definitions of international and standard organisations [33]:

- *ISO* 9126: "Software quality characteristic is a set of attributes of a software product by which its quality is described and evaluated".
- German Industry Standard DIN 55350 Part 11: "Quality comprises all characteristics and significant features of a product or an activity which relate to the satisfying of given requirements".
- Philip Crosby: "Quality means conformance to requirements " [22].
- ANSI Standard (ANSI/ASQC A3/1978): "Quality is the totality of features and characteristics of a product or a service that bears on its ability to satisfy the given needs".
- *IEEE* Standard (*IEEE* Std 729-1983):
  - The totality of features and characteristics of a software product that bear on its ability to satisfy given needs, *i.e.,* conformance to specifications.
  - The degree to which software possesses a desired combination of attributes.

– The degree to which a customer or a user perceives that a software meets her composite expectations.

– The composite characteristics of a software that determine the degree to which the software in use will meet the expectations of the customer.

All these definitions give separate, complementary views on quality. We accept all these definitions because each present an interesting point of view or complement to other definitions. However, we organize, clarify, and standardize the large number of quality-related definitions to obtain a complete view on quality. Figure 2.1 presents our meta-model of relationships among requirements models and quality models, which defines a model of quality.

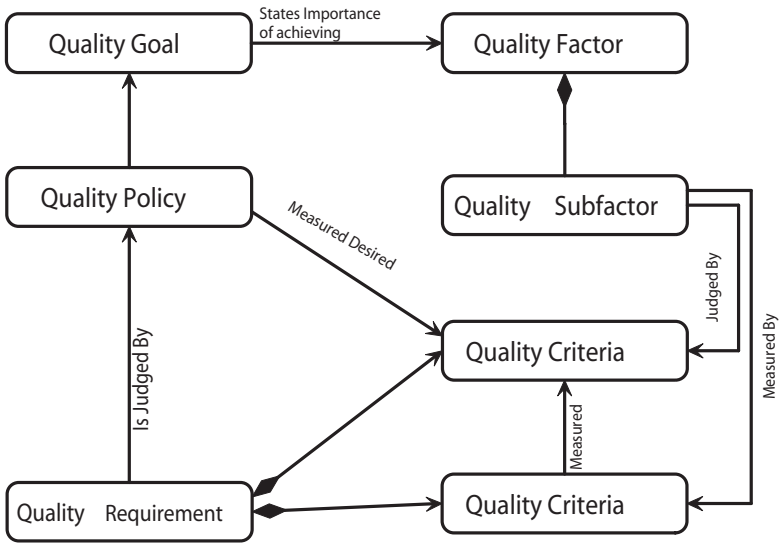

Figure 2.1: Relationships among requirements models and quality models

2.5 Quality Evaluation Tools

Evaluation of quality requires models to link measures of software artifacts with external, high-level, quality characteristics. First, we introduce the concept of quality model, then we present different elements related to quality models.

2.5.1 Quality Models

ISO/IEC 9126-1 defines a quality model as a "framework which explains the relationship between different approaches to quality" [47]. Most quality models decomposes in

hierarchical elements. An approach to quality is to decompose quality in factors, sub-factors, and criteria. Evaluation of a program begins with measuring each quality criteria with numerical value from metrics. Then, each quality sub-factors is assessed using their criteria. Finally, numerical value are assigned to quality characteristics from their quality sub-factors. Figure 2.2 presents a meta-model of the relationships among quality model elements. Appendix IV presents different definitions of quality models, which are used in software engineering literature.

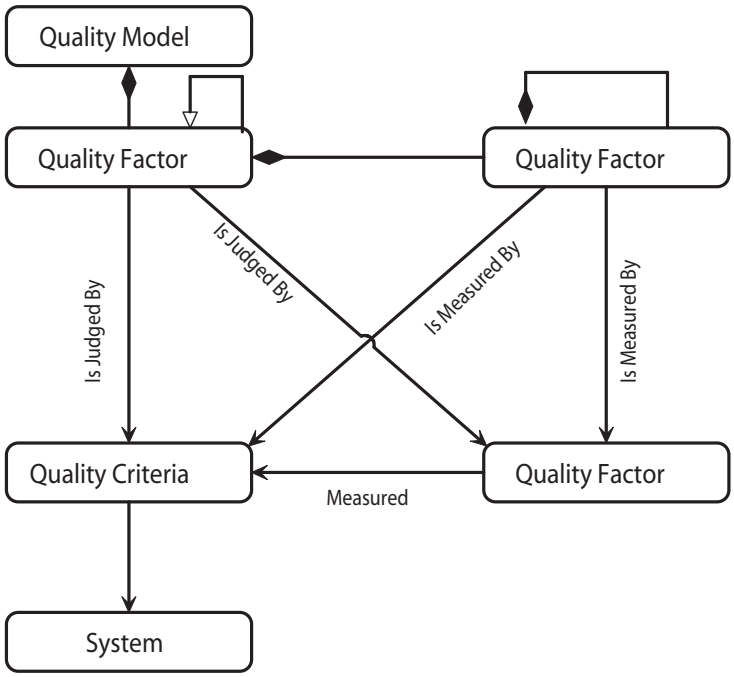

Figure 2.2: Relationship among quality model elements

2.5.2 Quality Characteristics

The typical objective of a quality factor is to characterize an aspect of the quality of a work product or a process [31]. Different terms are used for quality characteristics in various quality models [17], see Table 2.1.

2.5.3 Quality Sub-characteristics

Some characteristics can not be evaluated directly, they require an extra intermediate level to be computed. Elements of this intermediate level are sub-characteristics. For example, in Boehm's model (Figure IV.2) maintainability[3] relates to three sub-

[3]All the "-ility" are defined in Appendix II.

Quality Model	First Layer
Boehem Model	High-level Characteristic
McCall Model	Factor
ISO Model	Characteristic
IEEE Model	Factor
Dromey Model	High-level Attribute

Table 2.1: Different names for quality characteristics

Quality Model	Second Layer
Boehem Model	Primitive Characteristic
McCall Model	Criteria
ISO Model	Subcharacteristic
IEEE Model	Subfactor
Dromey Model	Subordinate Attribute

Table 2.2: Different names for quality sub-characteristics

characteristics: testability, understandability, and modifiability.

The typical objectives of a quality sub-factor are to [31]:

- Characterize a part of a quality factor.
- Further characterize an aspect of the quality of a work product or process.
- Help in defining the term "quality".

Table 2.2 presents the different terms which are used for quality sub-characteristic in various quality models [17].

2.5.4 Software Metrics

Software metrics are used to quantify software, software development resources, and software development processes. Some software characteristics are measurable directly (such as number of Lines of Code, better known as LOC), others characteristics can be inferred only from indirect measurements (*i.e.,* maintainability), and yet others are mostly related to human perception (*i.e.,* understandability is as dependent on the people as on program source code).

Software metrics can be classified into three categories [51]:

- Product metrics describe the characteristics of the product, such as size, complexity, design features, performance, and quality level.

- Process metrics can be used to improve software development and maintenance process (*i.e.,* effectiveness of defect removal during development, defect arrival, response time to fix defect).
- Project metrics describe characteristics of the project and its execution (*i.e.,* number of software developers, staffing pattern related to the life cycle of the software, cost, schedule, and productivity).

However, some metrics belong to multiple categories (*i.e.,* the in-process quality metrics of a project are both process metrics and project metrics). Moreover, a healthy metrics program focuses on much more than the measurement of programmer productivity. Indeed, the following areas of software development benefit from a well-planned metrics program [84]:

- Project management.
- Product quality.
- Product performance.
- Development process.
- Cost and schedule estimation.

2.5.5 Quality Metrics

"Software quality metrics are a subset of software metrics that focuses on the quality aspects of the product, process, and project. In general, software quality metrics are more closely associated with process and product metrics than with project metrics" [51].

We need to specify quality metrics to evaluate the quality of a product. Each quality metric will provide a numerical value that can be scaled to measure a quality characteristic. Metrics must be complete and detailed sufficiently to be the firm foundation of a quality model.

2.5.6 Internal Quality

N. Bevan defined the internal quality as a characteristic "which is measured by the static properties of the code, typically by inspection (such as path length)" [10] and K. Beck states that internal quality is quality as measured by the programmers [9, 21]

2.5.7 External Quality

By the definition of N. Bevan, external quality is defined as a characteristic "which is measured by the dynamic properties of the code when executed (such as response time)" [10]. More generally, K. Beck defines external quality as the quality which is measured by the customer [9, 21].

2.5.8 Quality in Use

ISO/IEC 9126-1 defines quality in use as "the user's view of quality. Achieving quality in use is dependent on achieving the necessary external quality, which in turns is dependent on achieving the necessary internal quality" [47], "which is measured by the extent to which the software meets the needs of the user in the working environment (such as productivity)" [10]. Quality in use decomposes into four characteristics [47]:

- Effectiveness
- Productivity
- Safety
- Satisfaction

2.6 Quality and Maintenance

A software process is defined as all the activities and work products which are necessary to develop a software system. These activities are [1]:

- Software specification (requirements , functionality and constraints).
- Software development (design, implementation).
- Software validation (ensure that the software meets the customer needs).
- Software evolution (evolve to meet changing customer needs).

In the past decade with the adoption of object oriented programming, facilities in software tools, the major part of software development is allocated with software maintenance. Maintenance is expansive and tedious because of the difficulty to predict maintenance effort, to allocate and to schedule appropriate personnel and resources[4].

[4]Maintenance cost of object-oriented programs during past decade grow more than some 60% in 1997 [73, 76] and some 80% in 2005 of the overall cost of programs.

To reduce the cost of software development in the future, we need to reduce the cost of maintenance. High quality software is the best way to reduce the software maintenance cost. There is two way to assessing the software qualities:

- Rules to evaluate the quality at the design level.
- Tools to evaluate the quality at the source code level.

2.7 Maintenance

As *IEEE* is defined, Software Quality could be one of the most important part of software maintenance.

Software Maintenance is defined as the totality of activities required to provide cost-effective support to a software system [2].

Software maintenance cost is related to following factors [61]:

- Configuration management (Software Configuration Management).
- Management (Software Engineering Management).
- Improvement (Software Engineering Process).
- Software Quality (Quality assurance, verification, validation).
- Project Management.

Prediction of maintenance effort mainly depends on the quality of the programs to maintain. Assessing the quality of programs require quality models to link software artifacts with quality characteristics.

Software maintenance could be compare with house maintenance, the quality of building's material and framework, could reduce the price of its maintenance.

2.8 Essentials of Measurement

Measurement is essential to any engineering discipline and, thus, software engineering. Scientific measurement pursues the following steps:

- Formulation of a theory or model is the first step of scientific measurement.
- Definition and collection of measures quantifying key elements which are related to the theory.

- Analysis of the collected data supports the evaluation of the accuracy of the original theory, and the effectiveness of the measures themselves.

Figure 2.3 illustrates the relationships of measures between from theory and practice [85].

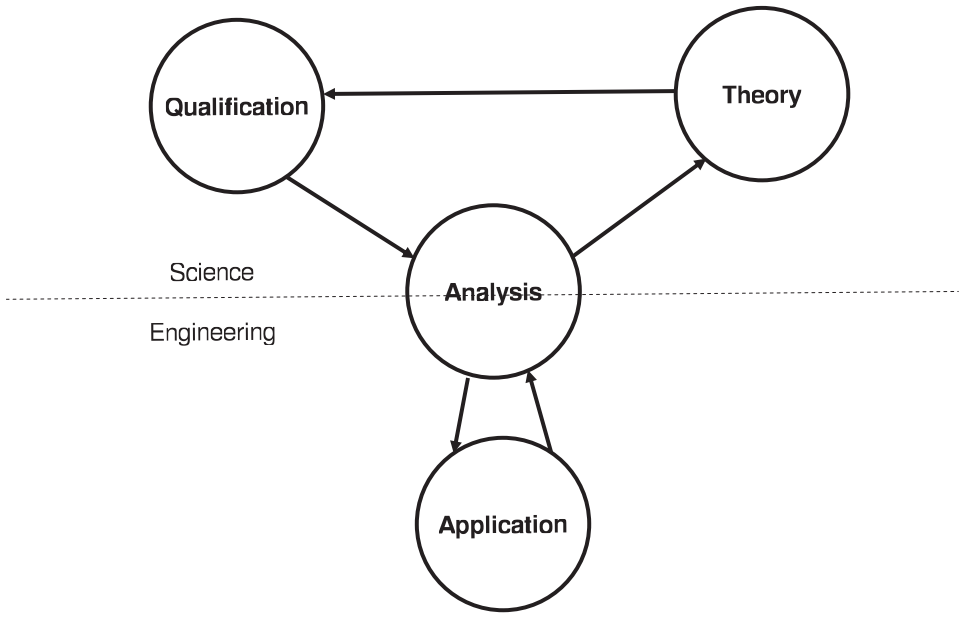

Figure 2.3: The Evolution of Measures [85]

In a book on software development and reality construction [34], the authors acknowledge the dichotomy—in traditional science—between observer and observation and the view of human needs as being *outside* of the realm of scientific enquiry. They regret this view and emphasise theirs:

> An important aspect of computer science is that it deals with *creating reality*: The technical reality of the programs executed on the computer, and the condition for the human reality which unfolds around the computer in use. Therefore, the conceptual categories "true" and "false" [that computer science] relies on are not sufficient in themselves. We have to go beyond them by finding categories for expressing *the felicity of our choices*, for distinguishing "more or less suitable" as we proceed in making distinctions and decisions in

communicative design processes. This is essential for dealing with quality in software development and use [34].

For example, one observer could record, over a period of a year, the Sun rises. Then, she could state a law generalising these events with the following "Everyday, the Sun rises". Finally, she could develop a theory of the Earth spinning and orbiting around the Sun, thus explaining the laws and the observations.

In other fields of science, such as physics, the discovery of relationships among artifacts follows the scientific method of observations, laws, and theories. [7,40]. This method consists in recording observations on some facts, then stating laws generalising the observations, finally developing a theory able to explain the laws and thus the observations.

The scientific method begins with observations. We believe that the field of software quality is so far limited by the lack of concrete and consensual facts on the quality of software. Facts in software engineering, in particular about quality, remain too complex because of the many interactions between software artifacts and of the discreet nature of software.

We must find facts in software engineering pertaining to software quality. We believe that design patterns, in particular their design motifs (solutions), could be such facts. Indeed, design motifs are concrete artifacts in programs and consensual in the quality characteristics they bring.

2.9 Conclusion

We concur that quality is a subjective value and is based on human perceptions because, at the end, the consideration of software quality is related to human needs. However, this view has not yet gained wide acceptance among the scientific community interested in software quality, because it makes it all-the-more difficult to define and to use quality models. In particular, this human-centric view is perceived to threaten previous and current context-insensitive researches on quality, even though it is a natural complementary extension only. We believe that this lack of acceptance leads to several issues when dealing with entire tools we have for assessing software quality. Also, quality is mainly measured at the source code level, without considering the programs architectures. In the rest of this work, we attempt to design a quality model incorporating the user's needs and taking in account the programs architectures.

CHAPTER 3

QUALITY EVALUATION

Based on our passed experience in assessing software quality, we state some open issues related to software quality by extend the classical view of software quality models and software metrics. we get trough the following questions:

- What are quality models and how should they be adapted and used?
- How could software metrics be misused to evaluate software quality?
- How could we make a better evaluation by improving our view from software metrics?

Thus, we propose some possible solutions to the mentioned open issues:

- Modifying software quality models so that characteristics and sub-characteristics are more meaningful to their users;
- Considering the dynamic behavior of the software during its execution;
- Using design patterns as high level building blocks to asses the software design.

All these solutions can bring significant improvements in the evaluation of the quality of programs. Also, we review software evaluation tools, some problems, and solutions as well as introducing new methods for assessing the software quality by using intermediate levels between quality characteristics and metrics to make the process of software quality's evaluation more efficient.

3.1 Open Issues

Software metrics and quality models play a pivotal role in measurement of software quality. A number of well-known quality models and software metrics are used to build quality software both in industry and in academia. However, during our research on measuring software quality using design patterns, we faced many issues related to existing software metrics and quality models. We discuss some of these issues and present our approach to software quality evaluation.

3.1.1 Human Estimation

A person can look at source code or software products and judge their performance and their quality. Human evaluation is the best source of measurement of software quality because it is a person who will deal with the software at the end. However, human assessment is not always possible nor desirable. First, the size of the software may be too big for one person to comprehend entirely. Second, as a person gets acquainted with a program source code, her evaluation of its quality may become biased. Moreover, the knowledge of quality is embedded in a person experience and hardly reusable.

3.1.1.1 Problem

Different taste, different value. Often, software evaluation of one person cannot be expanded as an acceptable evaluation for other people because different people have different views on quality. For example, just listen to other people advising for choosing an operating system or a wordprocessor...

Assessing the quality of software by your own is not practical. It is impossible that everybody has the knowledge and the ability to evaluating software performance and quality. In addition it is a very hard and time consuming task.

3.1.1.2 Solution

Birds of a feather flock together. We must categorize the people who deal with software at different level by considering their need for software quality, and then we can create tailored models for each group, or range of values which are acceptable for similar people. Categorisation could be performed using simple categories and questionnaire or with more sophisticated means, such as attitude analyses. For example, end users mostly have similar ideas about quality of software, but these ideas maybe different from those of people who deal with the maintenance of the same software.

3.1.2 Software Metrics

To the best of our knowledge, instead of using human estimation, software metrics are the only mechanized tools for assessing the value of internal attributes [47].

Software metrics are defined as "standard of measurement, used to judge the attributes of something being measured, such as quality or complexity, in an objective manner" [60], but subjective measurement of quality comes from human estimation.

3.1.2.1 Problem

Evaluation of software code is not enough. We believe that considering a source code with no regard for its execution is the same as considering the body without its sprit. Well-known metrics are just computing size, filiation, cohesion, coupling, and complexity. These internal attributes are related to code but the quality of a software does not depend on its code only: Acceptable quality in code evaluation does not guarantee performance and quality of software in execution with respect to the user's expectation.

Acceptable value for metrics evaluation. With different views of quality, it is hard to find a numerical value for quality which could be acceptable by all the people. Also, having different views affects software categorization in certain classification by considering the numerical value as the only parameter on software evaluation.

3.1.2.2 Solution

Code without value of execution is not valuable. The value of software metrics must be modified by runtime values for better results. Also, using a good structure and patterns (such as design patterns [38]) in the software design and resulting architecture could increase the software quality. Thus, we want to consider the architectural quality of software by considering the use of design patterns in the software architecture.

3.1.3 Quality Model

A quality model is a schema to better explain of our view of quality. Some existing quality models can predict fault-proneness (see for example [16]) with reasonable accuracy in certain contexts. Other quality models attempt at evaluating several quality characteristics but fail at providing reasonable accuracy, from lack of data mainly.

We believe that quality models must evaluate high-level quality characteristics with great accuracy in terms well-known to software engineers to help maintainers in assessing programs and thus in predicting maintenance effort.

Such quality models can also help developers in building better quality programs (i.e., with improved quality characteristics wrt. similar programs or previous versions) by exposing the relationships between internal attributes and external quality characteristics clearly.

We take a less "quantitative" approach than quality models counting, for example, numbers of errors per classes and linking these numbers with internal attributes. We favour a more "qualitative" approach linking quality characteristics related to the maintainers' perception and work directly.

3.1.3.1 Problem

Sub-characteristics unequally impact quality characteristics. In the literature, quality models define the relation between quality characteristics and sub-characteristics. However, the impacts of quality sub-characteristics on characteristics are not equivalent. For example: Adaptability and Instalability are two sub-characteristics related to Portability. The question is: If we assess the value of Adaptability as A and the value of Instalability as B, then is the value of Protability equals to A+B or $\frac{2}{3}A + \frac{1}{3}B$ or another function. Other methods to combine quality characteristics exist (such as Bayesian nets for example). However, to the best of our knowlegde, there exists no quality model so far that is consensual and agreed upon by the research community.

Main concept of quality is missing: In 384 BCE, Aristotle, as a scientist, knew all about medicine, philosophy... In 2005 CE, the concept of quality is the same as science in the age of Aristotle: Quality does not distribute in specific part, when we talk about software quality, we talk about assessing entire items which are part of the concept of quality.

3.1.3.2 Solution

Coefficient. Quality as an objective value is dependent on sets of software attributes and customer's requirements. These attributes are different level of characteristics and sub-characteristics in models of quality, but the relation and impact of each characteristic and sub-characteristic should be distinguished. Models can be made more meaningful for different persons by using coefficients which relate characteristic and sub-characteristic. These coefficients define the relationships, in a quality model, among characteristics and sub-characteristics. They allows emphasising the relationships important to a given person in a given person.

Jack of all trades and master of none. Assessing all the attributes related to software quality represent an important work. We extend quality models by defining a subject (super-characteristic) to focus on as base concept in quality. The super-characteristic describe the context of the model.

3.2 Our approach to Software Quality Evaluation

We introduce 3 steps needed to apply our approach to software quality evaluation, which solves some of the open issues. The following steps highlight the main ideas to implement software quality evaluation while considering human requirements and the programs architectures using design patterns [38].

3.2.1 Step 1: Choosing Category of People.

We must choose at least a person from the category of people for which our software evaluation will be implement, for example: Programmers, End-user ... In our case, we consider software developers or maintainers as the users of the quality model.

3.2.2 Step 2: Building a Quality Model.

The process of building a quality model decomposes in two main tasks generally:

- Choosing a super-characteristic.
- Choosing and organising characteristics related to super-characteristic.

In our case study, we consider design patterns especially as bridge between internal attributes of programs, external quality characteristics, and software engineers.

3.2.3 Step 3: Software Evaluation.

Finally, we apply our quality model on programs through the use of metrics related to the chosen characteristics.

3.3 Conclusion

In the following, we detail our approach to building a software quality model. Before detailing our model, its characteristics, and our building methodology. We introduce a study of design patterns and quality characteristics, which we need to relate “good” architectural practices (design patterns) with quality characteristics.

CHAPTER 4

DESIGN PATTERNS AND QUALITY EVALUATION

4.1 Brief History

Design patterns [38] are defined as high level building blocks that promote *elegance* in software by offering proven solutions to common problems in software design. Design patterns convey the experience of software designers.

Our work is at the conjunction of two fields of study: Quality models, on the one hand, design patterns and architectural models, on the other hand.

"During the late 1970s, an architect named Christopher Alexander carried out the first known work in the area of patterns, Alexander and his colleagues studied different structures that were designed to solve the same problem. In 1987, influenced by the writings of Alexander, Kent Beck and Ward Cunningham applied the architectural pattern ideas for the software design and development. In 1994, the publication of the book entitled Design Patterns: Elements of Reusable Object-Oriented Software on design patterns by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides explained the usefulness of patterns and resulted in the widespread popularity for design patterns" [57].

Design patterns are recurring solutions to design problems in real-world software development. Design patterns are about design and interactions among objects, as well as providing a communication platform concerning elegant, reusable solutions to commonly encountered programming challenges [24, 38].

Design patterns provide *good* solutions to design problems, which maintainers can use in the evaluation of the quality characteristics of program architectures naturally. Indeed, "[a]ll well-structured object-oriented architectures are full of patterns" [38, page xiii]. Also, design patterns provide a basis for choosing and for organising external quality characteristics related to the maintenance effort.

Design patterns are a body of literature to help software developers resolve recurring problems encountered throughout all of software development. Patterns help create a shared language for communicating insight and experience about these problems and their solutions. Formally codifying these solutions and their relationships captures the body of knowledge which defines our understanding of good architectures that meet the needs of their users.

"Forming a common pattern language for conveying the structures and mechanisms of our architectures allows us to intelligibly reason about them. The primary focus is

not so much on technology as it is on creating a culture to document and support sound engineering architecture and design" [4].

4.2 Design Patterns and Quality

Design patterns are defined as high level building blocks which promote elegance in software by increasing: flexibility, scalability, usability, reusability, and robustness. We try to evaluate them to find out how much they could increase software quality to be help for decrease the cost of software maintenance. However, there is also some evidences that design patterns do not intrinsically promote quality, see for example [90]. Yet, these evidences were accumulated in a software project where the use of design patterns went out of control, because of the developers' eagerness to use these good practices.

4.2.1 Bart Wydaeghe *et al.*

In the following there is the result of their experience on the using design patterns over a big software with following characteristics [94]:

Software Specification.

- 5.000 lines of code(out of system classes).
- Contained 173 classes.
- Developed and installed on different platforms
- used *Sparc 2.5*, *PC*'s with *Windows 95* and *Windows NT*.
- Chosen Java as programming language.

Design Patterns. Following design patterns are used for implementing the editor.

- Model-View-Controller
- Observer
- Visitor
- Iterator
- Bridge
- Façade
- Chain of Responsibility

Design Patterns	*Software Quality Characteristics*				
			Understandability		
	Modularity	*Flexibility*	*Expert*	*Layman*	*Reusability*
MVC	*Good*	*Good*	*Good*	*Bad*	*Excellent*
Observer	*N/A*	*Good*	*N/A*	*Bad*	*Good*
Visitor	*N/A*	*Good*	*N/A*	*Bad*	*Good*
Iterator	*N/A*	*Good*	*N/A*	*Bad*	*Good*
Bridge	*Good*	*Good*	*Good*	*Good*	*Excellent*
Façade	*Good*	*Good*	*Good*	*Good*	*Excellent*
Chain of Responsibility	*Bad*	*Good*	*N/A*	*Good*	*N/A*

Table 4.1: Evaluation of OMT-Editor by using design patterns

Result. Table 4.1 is result of evaluation of OMT-Editor for different design patterns. It shows that design patterns do not equally promote quality and that their quality depends on the context.

4.2.2 Wendorff [90]

Wendorff presents the result of a practical experience during a large commercial project that has taken place at one of Europe's largest service companies between 1994 and 2000. The project in question caused an effort of several hundred man–years, with more than 50 programmers involved at peak times. The code size is 1,000 KLOC (800 KLOC C++ and 200 KLOC PUSQL (with 1 KLOC = 1,000 lines of code)).

Cost of Removing the Patterns.

- **Proxy Pattern** "The removal of a proxy pattern in presence of complex pre-processing or post-processing, proved to be very difficult and needed careful attention to side effects. In one subsystem with about 3000 lines of code we removed 3 out of 7 proxy patterns altogether, leading PO a reduction of 200 lines of code".

- **Observer Pattern** "The removal of the MVC architecture would have amounted to a completely new design and implementation of the graphical user interface and was therefore out of the question".

- **Bridge Pattern** "Removing such a bridge pattern from code is usually a straightforward merger of two classes into one class. In one subsystem we removed 2 out of 3 bridge patterns with an economy of 190 out of 1400 lines of code".

- **Command Pattern** "... then the requirements changed. The idea to read the sequences of operations from a database was dropped. At that stage the command pattern architecture had developed into a very complex software artefact with loads of unnecessary and complicated features that was intricately intertwined with other parts of the software architecture. Even the original designers themselves freely admitted that their solution was completely over the top. In this situation a full re-engineering of the failed command pattern architecture would have been desirable, but because of its complexity that was no longer a viable option".

Comment. The author concludes that design patterns do not improve a program architecture necessarily. Indeed, architecture can be over-engineered [53] and the cost of removing design patterns is high. The author does not link his study with any quality model.

4.2.3 Other Authors

Briand and Wüst [16] present a detailed and extensive survey of quality models. They classify quality models in two categories: correlational studies and experiments. Correlational studies use univariate and multivariate analysis, while experiments use, for examples, analysis of variance between groups (ANOVA). To the best of our knowledge, none of the presented quality models attempts to assess the architectural quality of programs directly. They all use class-based metrics or metrics on class couples.

Wood *et al.* [92] study the structure of object-oriented C++ programs to assess the relation between program architectures and software maintenance. The authors use three different methods (structured interviews, survey, and controlled experiments) to conclude that the use of inheritance in object-oriented programs may inhibit software maintenance. Design patterns intensively use inheritance.

Harrison *et al.* [42] investigate the structure of object-oriented programs to relate modifiability and understandability with levels of inheritance. Modifiability and understandability cover only partially the quality characteristics that we are interested in. Levels of inheritance are but one architectural characteristic of programs related to software maintenance.

Wydaeghe *et al.* [94] assess the quality characteristics of the architecture of an OMT editor through the study of 7 design patterns (Bridge, Chain of Responsibility, Facade, Iterator, MVC, Observer, and Visitor). They conclude on flexibility, modularity, reusability, and understandability of the architecture and of the design patterns. However, they do not link their evaluation with any evaluative or predictive quality model.

Tatsubori *et al.* [86] in their experiment used compile-time MOPs to provide a general framework for resolving implementation problems of design patterns. Some programs written according to design patterns are too complicated and error-prone and their overall structure is not easy to understand. They conclude that programs with comments are more understandable than using the design patterns for maintenance.

4.3 Quality Evaluation of Design Patterns

We believe that the problem of quality with design patterns comes both from the design patterns themselves and from their misuse. Unfortunately, little work has attempted so far to study the quality characteristics of design patterns rigorously.

Design patterns claim [38] to make object oriented designs more:

- Flexible.
- Elegant.
- Reusable.

Indeed, Gamma *et al.* state that design patterns "make object-oriented more *flexible*, *elegant* and ultimately *reusable*" and "[D]esign patterns help you chose design alternatives that make a system reusable and avoid alternatives that compromise reusability. Design patterns can even *improve the documentation* and *maintenance* of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent", [38, page 2].

Elegancy is defined as maximizing the information delivered through the simplest possible interface. Issues of elegance in software are reflected to robustness, scalability, flexibility, and usability [35].

- Robustness.
- Scalability.
- Flexibility.
- Usability.

Thus, we can consider that design patterns claim to increase the following quality characteristics:

- Flexibility.

- Reusability.
- Robustness.
- Scalability.
- Usability.

Some quality characteristics are supposed to increase when we use design patterns to develop software product. To assess the truthfulness of this assumption, we need a quality model that contains these quality characteristics and which relates metrics with these quality characteristics to make them measurable.

4.3.1 Model Definition

To define attributes and metrics for our model[1], we start with standard definitions from *IEEE* and *ISO/IEC* and expand them with other models.

- Usability: *ISO/IEC* defines usability as part of the quality characteristics related to the following attributes:
  - Understandability.
  - Learnability.
  - Operability.

  McCall's model (in [33]) defines the usability as:
  - Operability.
  - Training.
  - Communicativeness.

  To cover the understandability attributes, Boehm's model defines understandability as a characteristic related to:
  - Structuredness.
  - Conciseness.
  - Legibility.

[1] We consider a hierarchical model, because it is more understandable and also most of the standard models are hierarchical.

- Reusability: McCall's model defines reusability as a characteristic related to the following attributes:
  - Software system independence.
  - Machine independence.
  - Generality.
  - Modularity.
- Flexibility: McCall's model defines flexibility as a characteristic related to the following attributes:
  - Self Descriptiveness.
  - Expendability
  - Generality.
  - Modularity.
- Scalability: Smith and Williams are defined scalability as "the ability of a system to continue to meet its response time or throughput objectives as the demand for the software functions increases" [82]. We consider the processing level and software performance as characteristics related to scalability.
- Robustness: Firesmith [32] defines robustness as a characteristic related to:
  - Environmental tolerance.
  - Error tolerance.
  - Failure and fault tolerance.

We combine the characteristics and the sub-characteristics together and we propose a model for evaluating these characteristics, in Figure 4.1. Some of the characteristics in our model could be measured according to different view points, for example learnability or understandability have different definitions from the user's or developer's view. We always consider the developers' or maintainers' point of view.

We must consider that design patterns are only (mostly) defined for object oriented programming [38] and take consideration that object-oriented programming language originated around 1987 [75]. McCall's model (1976-7) and Boehm's model (1978) never had a view of object oriented programming thus it is necessary to review each characteristics, attributes, and metrics defined in our model and to redefine them by considering object oriented programming. We consider characteristics and sub-characteristics related to object oriented programming by adapting our model, see Figure 4.2.

4.4 Conclusion

In this chapter, we put in perspective design patterns and quality models and showed that no previous quality model take in account the architecture of programs or "good" practices, such as design patterns. We proposed a quality model for evaluating the quality of programs using patterns. This model include several quality characteristics and sub-characteristics. We now detail metrics used to measure this characteristics.

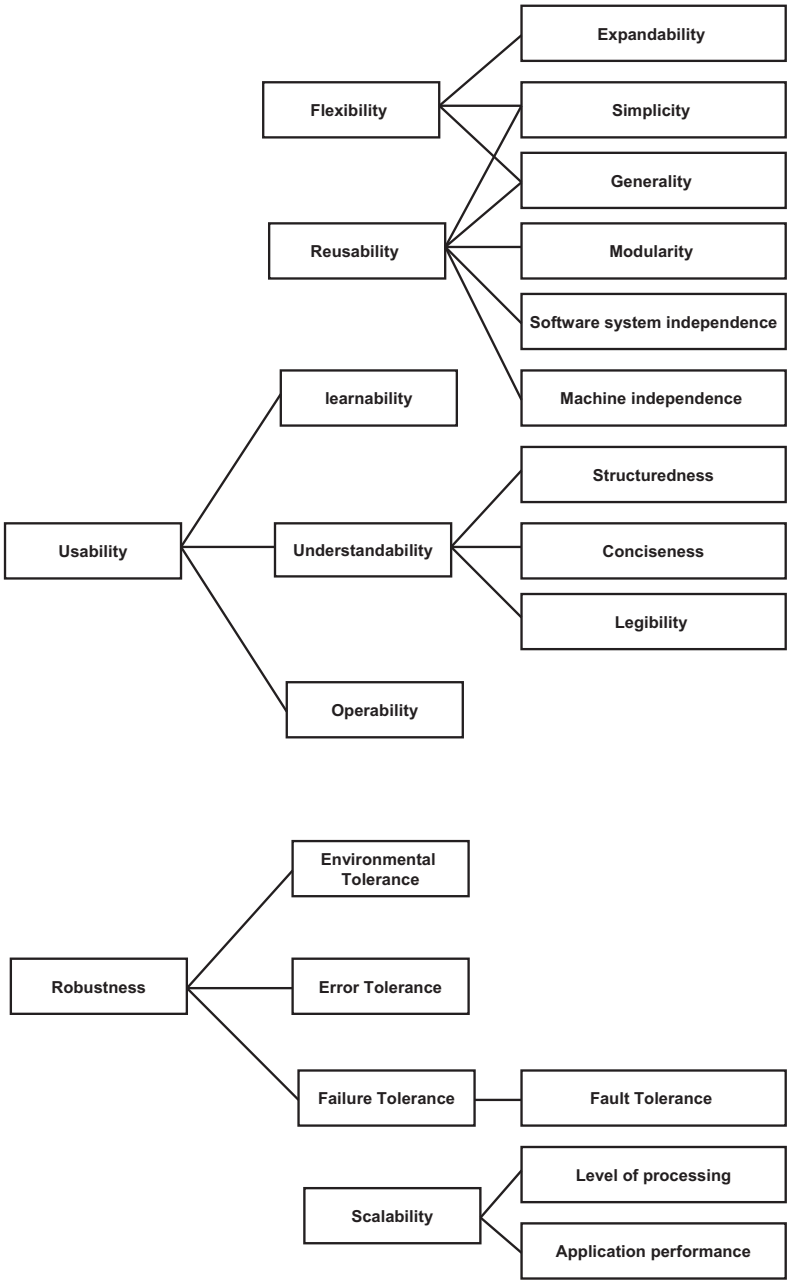

Figure 4.1: Model for assessing the quality characteristics of design patterns.

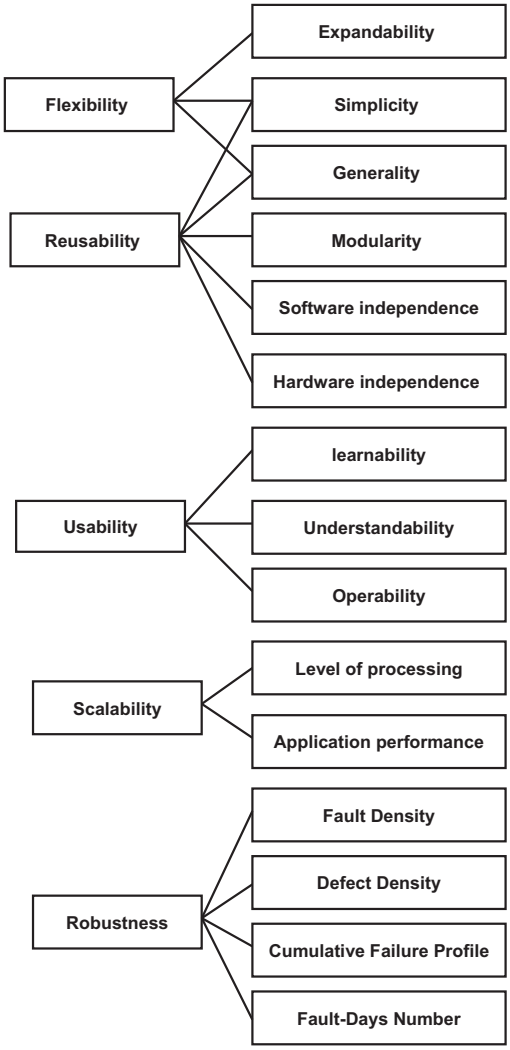

Figure 4.2: Model for accessing the quality in software implemented using design patterns.

CHAPTER 5

METRICS FOR QUALITY EVALUATION OF DESIGN PATTERNS

Software quality metrics are a subset of software metrics that focus on the quality characteristics of software products, processes, and projects. In this chapter, we present some quality metrics based on *IEEE* Software Quality [45] and *ISO* 9126 [48] which could be used directly or indirectly to assess characteristics and sub-characteristics in our model, see Figure 4.2. In the following, we define the quality characteristics in our model and associate each of these characteristics with metrics.

Expandability: National Institute of Standards and Technology [77] define expendability as attributes for assessing the adaptability in quality of code as follow:

- processing independent of storage [20, 77]:
  $\frac{\text{(number of modules whose size constraints are hard}-\text{coded)}}{\text{(total number of modules with such size constraints)}}$
  The module necessities is:
  - Independent of storage size, buffer space, array sizes, etc.
  - Provided dynamically, e.g., array sizes passed as parameters.
- Percentage of uncommitted memory [20, 77]:
  $\frac{\text{(amount of uncommitted memory)}}{\text{(total memory available)}}$
- Percentage of uncommitted processing capacity [20, 77]:
  $\frac{\text{(amount of uncommitted processing capacity)}}{\text{(total processing capacity available)}}$

Simplicity: Simpliciy is the opposite of complexity, which is measured by considering the following elements:

- Static complexity measures the complexity of modules as a network, related with design transaction analysis. Static complexity is calculate as:
  E: The number of edges, indexed by $i = 1, \ldots, E$.
  N: The number of modules, indexed by $j = 1, \ldots, N$.
  $C = E - N + 1$.

- Generalized Complexity measures the complexity as represented with a network of modules and of used resources. Generalized complexity is calculated as:
  K: The number of resources, indexed by $k = 1, \ldots, K$.
  c_i: Complexity for program invocation and return along each edge e.
  r_{ki}: 1 if the k^{th} resource is required for the i^{th} edge, 0 else.
  d_k: Complexity for the allocation of resource k.
  $C = \sum_E^{i=1} \left(C_i + \sum_K^{k=1} d_k * r_{ki} \right)$.

- Dynamic Complexity measures the complexity during program execution. Dynamic complexity is calculated with static complexity formula at different point of the execution.

- Fan-in / fan-out: This metric is defined as the number of entries / exits per module and is used for evaluating data encapsulation especially.
  e_i: The number of entry points for the i^{th} module.
  x_i: The number of exit points for the i^{th} module.
  $m_i = e_i + x_i$.

- Data flow complexity is defined by the set of attributes related to the complexity of data.
  lfi: Local flows into a procedure.
  lfo: Local flows from a procedure.
  datain: Number of data structure that are accessed by a procedure.
  dataout: Number of data structure that are updated by a procedure.
  length: Number of real statements in a procedure source code (excluding comments).

  - Number of live variables:
    lv_i, number of live variables in the i^{th} executable statement.
    n: Total number of executable statements.
    m: Total number of modules.
    $\overline{LV} = \frac{\sum_n^{i=1} lv_i}{n}$.
  - Average number of executable statements:
    $\overline{LV}_{program} = \frac{\sum_m^{i=1} \overline{lv_i}}{m}$
  - Variable spans:
    sp_i: Number of statements between two references to the same variable.
    n: Total number of statements.
    $\overline{SP} = \frac{\sum_n^{i=1} sp_i}{n}$.

  - Average spans size of program:
    $\overline{SP}_{program} = \frac{\sum_{n}^{i=1} \overline{sp_i}}{n}$.

- Code Complexity:

  - Number of Decision: number of I/O variables, conditional, and loop control statements.
  - Cyclomatic complexity: computation of cyclomatic complexity is based on the flow-graph of the module:
    v: Complexity of graph.
    e: Number of edges (nodes between programs flow).
    n: Number of nodes (sequential group).
    S: Number of splitting nodes (sequential group).
    DE_i: Number of decisions for the i^{th} module or number of conditions.
    * Connected graph: If a connected graph is built then:
      $v = e - n + 1$.
      Else:
      $v = e - n + 2$.
    * Splitting nodes:
      $v = S + 1$
    * Sum of v: the cyclomatic complexity for a multi-modules program can be measured by summing values of v for individual modules.
      $v_{program} = \sum_{m}^{i=1} v_i = \sum_{m}^{i=1} DE_i + m$.
  - Average nesting level:
    L_s: Nesting level of statements, defined as the numerical value for the level of a statement (higher level are defined for main loop, followed by module and statements containing loops, conditional clauses...).
    S_t: Total number of statements.
    Average Nesting Level $= \frac{\sum L_s}{S_t}$.
  - Executable lines of code: Complexity can be measured by counting the number of executable lines of code per module.
  - Halstead metrics: These metrics measures the properties and the structure of programs. They provide measures of the complexity of existing software, predict the length of a program, and estimate the amount of time an average programmer can be expected to use to implement a given algorithm.

These metrics compute the program length by counting operators and operands. The measure suggests that the difficulty of a given program can be derived, based on the below counts [45]:
n_{tr}: Number of unique operators.
n_{nd}: Number of unique operands.
N_{tr}: Total number of operators.
N_{nd}: Total number of operands.

* Program Vocabulary: $l = n_{tr} + n_{nd}$.
* Observed program length: $N = N_{tr} + N_{nd}$.
* Estimated program length: $\Upsilon = n_{tr}\,(log_2 n_{tr}) + n_{nd}\,(log_2 n_{nd})$.
* Jensen's estimator of program length: $N_F = log_2 n_{tr}! + log_2 n_{nd}!$.
* Program volume: $V = L\,(log_2 l)$.
* Program difficulty: $D = \left(\frac{n_{nd}}{2}\right)\left(\frac{N_{nd}}{n_{nd}}\right)$.
* Program level: $L1 = \frac{1}{D}$.
* Effort: $E = \frac{V}{L1}$.

Generality: Generality achieves a large reuse potential ability to handle any syntactically correct code collection [59]. Generality calculated is size of the application domain because the most important thing in reverse engineering could be the useful documentation for better identify common design and architectural. Some metrics related to generality define as Multiple usage metric [20, 77]:

- Multiple usage metric:
  M_{MU}: Number of modules referenced by more than one module.
  M_{total}: Total number of modules.
  $G_{MU} = \frac{M_{mu}}{M_{total}}$.
- Mixed function metric: M_{MF}: Number of modules that mix functions.
  M_{Total}: Total number of modules.
  $G_{MF} = \frac{M_{mf}}{M_{total}}$.
- Data volume metric:
  M_{DVM}: Number of modules that are limited by the volume of data.
  M_{Total}: Total number of modules.
  $G_{DVM} = \frac{M_{dvm}}{M_{total}}$.

- Data value metric:
  M_{DVL}: Number of modules that are data value limited.
  M_{Total}: Total number of modules.
  $G_{DVL} = \frac{M_{dvl}}{M_{total}}$.

- Redefinition of constants metric:
  M_{rc}: Number of constants that are redefined.
  M_{total}: Total number of modules.
  $G_{rc} = \frac{M_{rc}}{M_{total}}$.

Modularity: Modularity is measured using the relationships among the elements of a module. Higher strength modules tend to have lower fault rates, and tend to cost less to develop. Also, the modularity is greater for higher strength modules [18, 20, 77].

- Cohesion is evaluated by module using a scale from high for functional cohesion to low for coincidental cohesion:
  X: Reciprocal of the number of assignment statements in a module.
  Y: Number of unique function outputs divided by number of unique function inputs.
  $STRENGTH = \sqrt{(X^2 + Y^2)}$.

  More precisely, Module cohesion is defined as "how tightly bound or related its internal elements are to one another" [95]. Cohesion metrics apply to unit design cohesion, or module strength that refers to the relationships among the elements of a module. Design-level cohesion is defined as six relations between a pair of output components based on input-output dependence graph (IODG) representation [11, 52]:

  - Coincidental relation (R1):
    Two module outputs have neither dependence relationship with each other, nor dependence on a common input.
    $R_1(o_1, o_2) = o_1 \neq o_2 \wedge \neg(o_1 \rightarrow o_2) \wedge \neg(o_2 \rightarrow o_1) \wedge \neg\exists x\,[(x \rightarrow o_1) \wedge (x \rightarrow o_2)]$
  - Conditional relation (R2):
    Two outputs are c-control dependent on a common input, or one output has c-control dependence on the input and another has i-control dependence on the input.
    $R_2(o_1, o_2) = o_1 \neq o_2 \wedge \exists x \left[\left(x \xrightarrow{c} o_1\right) \wedge \left(x \xrightarrow{c} o_2\right)\right]$
  - Iterative relation (R3):
    Two outputs are i-control dependent on a common input.

$R_3(o_1, o_2) = o_1 \neq o_2 \wedge \exists x \left[\left(x \xrightarrow{i} o_1\right) \wedge \left(x \xrightarrow{i} o_2\right)\right]$

- Communicational relation (R4):
  Two outputs are dependent on a common input. One has data dependence on the input and the other has either a control or data dependence.
  $R_4(o_1, o_2) = o_1 \neq o_2 \wedge \exists x \left[\left(\left(x \xrightarrow{d} o_1\right) \wedge \left(x \xrightarrow{d} o_2\right)\right) \vee \left(\left(x \xrightarrow{q} o_1\right) \wedge \left(x \xrightarrow{q} o_2\right)\right)\right]$
  Where $p, q \in \{d, c, i\}$ and $p \neq q$
- Sequential relation (R5):
  One output is dependent on the other output.
  $R_5(o_1, o_2) = o_1 \neq o_2 \wedge [(o_1 \rightarrow o_2) \vee (o_2 \rightarrow o_1)]$
- Functional relation (R6):
  There is only one output in a module.
  $R_6(o_1, o_2) = (o_1 = o_2)$

Software Independence: For measuring the software independency, we need to considering these following options [20, 77]:

- Number of operating systems that could be compatible with the software can increase the software independency.
- Most of procedure is made of software system to use the utilities, libraries, and operating system and make the software more dependent on particular software environment. The degree of dependence on system software utilities have inverse related with software independency.
- "The usage of non-standard constructs or extensions of programming languages provided by particular compilers may impose difficulties in conversion of the system to new or upgraded software environments" [77]:
  $\frac{(\text{number of modules utilizing non-standard constructs})}{(\text{total number of modules})}$

Hardware Independence: The following options related to software independency, with evaluation of these software attributes we could find a numerical value for hardware independency:

- With considering the definition of open source programs as "Products based on open systems standards (Particularly the *ISO* open system interconnection (OSI) and *IEEE* POSIX) are beginning to replace reliance on proprietary

computing platforms" [58], this ability can increase the value of hardware independency.

- The dependency of software for using the programming languages and tools (like compilers, database management systems and user interface shells); with available implementation by other machines.
- The degree of using input/output references or calls, increase the hardware dependency. As follow we have a metrics to assess this software attribute [77]:
  $\frac{\text{(number of modules making I/O references)}}{\text{(total number of modules)}}$
- Code that is dependent on machine word or character size is another parameter that makes the software more dependence on machines hardware. As follow we have a metrics to have a numerical value for this attribute [77].
  $\frac{\text{(number of modules not following convention)}}{\text{(total number of modules)}}$

Learnability: Learnability is most of the human related part of usability, the ISO/IEC 9126-1 [48, Part 1 page 9] measure the learnability with considering the suitability as internal metrics for learnability.
ISO/IEC 9126-1 defined the suitability metric as follow[1] [48]:

$$X = 1 - \frac{\text{number of function in which problems are detected in evaluation}}{\text{number of functions checked}}$$

$$X = 1 - \frac{\text{number of missing functions detected in evaluation}}{\text{number of functions described in requirement specifications}}$$

$$X = 1 - \frac{\text{number of incorrectly implemented of missing functions detected}}{\text{number of functions described in required specifications}}$$

$$X = 1 - \frac{\text{number of functions changed during development life cycle phases}}{\text{number of functions described in required specifications}}$$

- Coupling is a measure of the degree to which modules share data. A lower coupling value is better [77].
  M_j: Sum of the number of input items shared between components i and j.
  Z_i: Average number of input and output items shared over m components with component i.
  n: Number of components in the software product.
  $Coupling = \frac{\sum_n^{i=1} Z_i}{n}$.
  $z_i = \frac{\sum_m^{j=1} M_i}{m}$.

Understandability in Design: This measure is used to determine the simplicity of the detailed design of a software program it uses the following primitives:

[1] X is acceptable between 0 and 1. If X is closer to 1, Operability is better.

- Number of nodes (sequential groups of program statements).
- Number of edges (program flows between nodes).
- Number of splitting nodes (nodes with more than one edge emanating from it).
- Number of regions (areas bounded by edges with no edges crossing).

The values determined for the primitives can be used to identify problem areas within the software design [45].
D_1: Design organized top down (Boolean).
D_2: Module dependence $= \frac{\text{Number of modules dependent on the input or output}}{\text{Total number of modules in the program}}$.
D_3: Module dependent on prior processing $= \frac{\text{Number of modules dependent on prior processing}}{\text{Total number of modules in the program}}$.
D_4: Database size $= \frac{\text{Number of non}-\text{unique database elements}}{\text{Number of database elements}}$.
D_5: Database compartmentalization $= \frac{\text{Number of database segments}}{\text{Number of database elements}}$.
D_6: Module single entrance/Single exit.
W_i: Weight given to thei^{th}derived measure.
$DSM = \sum_{6}^{i=1} W_i D_i$.

Understandability in Code: Understandability in outlook of programming is the characteristic by direct related with program code. Briand *et al.* [15] define understandability as:

- "To understand a method or class, we must know about the services the class uses".
- "Understandability is influenced by the number of services used. It should not matter if the server classes are stable or not".
- "To understand a class, we need to know the functionality of the services directly used by the class".
- "To analyze understandability, we do not need to account for polymorphism".

Operability: Operability defined with *ISO/IEC* 9126-1 [48, part-1 page-9] as the capability of software product to enable the user to operate and control it. Operability corresponds to controllability, error tolerance and conformity with user expectations. The metrics for measuring the operability define as follow[2] :
$X = \frac{\text{number of input items which check for valid data}}{\text{number of input items which could check for valid data}}$

[2]Our measured as X is acceptable between 0 and 1. If result is closer to 1, the Operability would be better.

$$X = \frac{\text{number of implemented functions which can be cancelled by the user}}{\text{number of functions requiring the precancellation capability}}$$
$$X = \frac{\text{number of implemented functions which can be undone by the user}}{\text{number of functions}}$$
$$X = \frac{\text{number of functions which can be customized during operation}}{\text{number of functions requiring the customization capability}}$$
$$X = \frac{\text{number of functions which can be customized}}{\text{number of functions}}$$
$$X = \frac{\text{number of functions having status monitoring capability}}{\text{number of functions that are required to have monitoring capability}}$$
$$X = \frac{\text{number of instances of operations with inconsistent behavior}}{\text{total number of operations}}$$
$$X = \frac{\text{number of implemented messages with clear explanations}}{\text{number of messages implemented}}$$
$$X = \frac{\text{number of interface elements which are self}-\text{explanatory}}{\text{total number of interface elements}}$$
$$X = \frac{\text{number of functions implemented with user error tolerance}}{\text{total number of functions requiring the tolerance capability}}$$

Scalability: To have a better definition for scalability, in the first we need some primitive definitions [49]:

- Speedup S: Measure the ratio of work increasing with change the number of possessors from 1 to k[3].
- Efficiency E: $E(k) = \frac{S(k)}{k}$.

Scalability measure as:
$\varphi(k_1, k_2) = \frac{E(k_2)}{E(k_1)}$

Level of Processing: The level of processing charaterises the software use of resources.

- Independence of storage: Metrics used to measure independence of storage are:
  M_{IS}: Number of modules which size constraints are hard-coded.
  M_{Total}: Number of modules with size constraints,
  $A_{IS} = \frac{M_{is}}{M_{total}}$
- Uncommitted memory: Metrics used to measure the adaptability with respect to percentage of uncommitted memory are:
  M_{UM}: Amount of uncommitted memory.
  M_{Total}: Total memory available.
  $A_{UM} = \frac{M_{um}}{M_{total}}$.
- Uncommitted processing capacity: Metrics used to measure the percentage of uncommitted processing capacity are:
  M_{UPC}: Amount of uncommitted memory.
  M_{Total}: Total memory available.
  $A_{UPC} = \frac{M_{upc}}{M_{total}}$.

[3]The best value for speedup is S(k) = k.

```
#include <stdio.h>
char    *msg="Hello World \n";
main()
{
    while(*msg)putchar(*msg++);
}
```

Figure 5.1: Program 1

```
#include <stdio.h>
main()
{
    puts("Hello World \n");
}
```

Figure 5.2: Program 2

Performance: Performance also called efficiency, is based on usage of CPU, RAM, and of I/O capacity. Thus, execution efficiency depends on:

- Non-loop dependency: Percentage of loop statements with non-loop dependent statements:
  M_{nl}: Number of modules with non-loop dependent statement in loops.
  M_{total}: Total number of modules.
  $E_{nld} = \frac{M_{nld}}{M_{total}}$.

- Compound expression: Repeated compound statements reduce efficiency:
  M_{rc}: Number of modules with repeated compound expressions.
  M_{total}: Total number of modules.
  $E_{rc} = \frac{M_{rc}}{M_{total}}$.

- Memory overlay[4]: Memory overlay creates overhead during processing and reduces the efficiency. Thus, the number of memory overlay defines a factor of software efficiency.

- Nonfunctional executable code: Nonfunctional executable code obscures efficiency. For example, Program 5.1 is more obscure than Programm 5.2:

  The calculation metrics define as:

[4]Execution is possible when entire program and data of process should be uploaded in physical memory, if the process is larger than memory, there is a technique called memory overlay: The idea of overlay is to keep in memory only those instructions and data that are needed at any given time [81]

M_{Nec}: Number of modules with nonfunctional executable code.
M_{Total}: Total number of modules.
$E_{Nec} = \frac{M_{nec}}{M_{total}}$.

- Inefficient coding decisions:
  M_{DS}: Number of modules with inefficient coding decisions.
  M_{Total}: Total number of modules.
  $E_{DS} = \frac{M_{ds}}{M_{total}}$.

- Data grouping: Efficiency decreases with complicated nesting and indices:
  M_{DG}: Number of modules with inefficient data grouping.
  M_{Total}: Total number of modules.
  $E_{DG} = \frac{M_{dg}}{M_{total}}$.

- Variable initialization: Initialization of variables during execution can reduce efficiency:
  M_{VI}: Number of modules with non-initialized variable declarations.
  M_{Total}: Total number of modules.
  $E_{VI} = \frac{M_{vi}}{M_{total}}$.

Storage efficiency depends on:

- Duplicate data definitions: Duplicate global data definitions consume space and reduce efficiency. A metrics for duplicate global data definition is defined by:
  M_{DDD}: Number of modules with duplicate data definitions.
  M_{Total}: Total number of modules.
  $E_{ddd} = \frac{M_{ddd}}{M_{total}}$.

- Code duplication:
  M_{DC}: Number of modules with duplicated code.
  M_{Total}: Total number of modules.
  $E_{DC} = \frac{M_{dc}}{M_{total}}$.

- Dynamic memory management: This is a boolean metric which considers the use of dynamic memory management. Value “true” indicates that allocated memory is released as needed, value “false” indicates that efficient memory use is not promoted.

- Requirements allocation: This is a boolean metric which evaluates the level of storage optimization by compiler or assembler, with value “true” for acceptable, and value “false” for non-acceptable.

Robustness: Champeaux (1997) define the flexibility as: "Flexibility also called Robustness, of a software system can be defined as the ability to perform even outside an intended application domain, or at least it has the feature that its functionality degrades graduality outside its domain." [25].

Robustness measure as following options [89]:

- Range of operating conditions (what can be done with it?)
- Amount of invalid behavior with valid input.
- Acceptability of behavior with invalid input.

Donald G. Firesmith measure robustness as following options [32]:

- Environmental tolerance: Degree to which an executable work product continues to function properly despite existing in an abnormal environment.
- Error tolerance: Degree to which an executable work product continues to function properly despite the presence of erroneous input.
- Failure tolerance: Degree to which an executable work product continues to function properly despite the occurrence of failures, where:
  - A failure is the execution of a defect that causes an inconsistency between an executable work product's actual (observed) and expected (specified) behavior.
  - A defect may or may not cause a failure depending on whether the defect is executed and whether exception handling prevents the failure from occurring.
  - A fault (also known as defect, bug) is an underlying flaw in a work product (*i.e.,* a work product that is inconsistent with its requirements, policies, goals, or the reasonable expectations of its customers or users). Defects are typically caused by human errors, and defects have no impact until they cause one or more failures.

  Failure tolerance includes the following quality sub-factor:

  - Fault tolerance: Degree to which an executable work product continues to function properly despite the presence or execution of defects.

As we saw we have a different taking from robustness but the relation definition and computation of robustness that could be close to main idea of using design patterns is considering the robustness computes as same measurement as reliability [45,69] in *IEEE* 982.1:

Before starting the definition of metrics related to robustness, we need to understand some primitive definitions those comes as follow [77]:
F = Total number of unique faults found in a given time interval resulting in failures of a specified severity level.
KSLOC = Number of source lines of executable code and non-executable data declarations in thousands.
D_i = total number of unique defects detected during the i_{th} design, or code inspection process, or the i_{th} life cycle phase.
I = total number of inspections to date.
KSLOD = In the design phase, the number of source lines of design statements in thousands.
Fd_i = Fault days for the i^{th} fault:

- Fault Density: This measure can be used to perform the following functions [45]:
  - "Predict remaining faults by comparison with expected fault density".
  - "Determine if sufficient testing has been completed, based on predetermined goals for severity class".
  - "Establish standard fault densities for comparison and prediction".

  To measure the fault density we need to follow the following steps:
  - Failure types might include input, output (or both) and user.
  - Fault types might result from design, coding, documentation, and initialization.
  - Observe and log each failure.
  - Determine the program fault(s) that caused the failure.
  - Classify the faults by type.
  - Additional faults may be found resulting in total faults being greater than the number of failures observed, or one fault may manifest itself by several failures.
  - Thus, fault and failure density may both be measured.

  - Determine total lines of executable and non-executable data declaration source code (KSLOC).
  - Calculate the fault density for a given severity level as:
    $F_d = \frac{F}{KSLOC}$

- Defect Density: "The defect density measure can be used after design and code inspections of new development or large block modifications. If the defect density is outside the norm after several inspections, it is an indication that the inspection process requires further scrutiny" [45].

  Defect density is measuring as following steps:

  - Establish a classification scheme for severity and class of defect.
  - For each inspection, record the product size and the total number of unique defects.
  - The defect density calculate in the design phase as [5] : $DD = \frac{\sum_{i=1}^{I} D_i}{KSLOD}$

- Cumulative Failure Profile: This is a graphical method used to [45]:

  - "Predict reliability through the use of failure profiles".
  - "Estimate additional testing time to reach an acceptably reliable system".
  - "Identify modules and subsystems that require additional testing".

  Establish the severity levels for failure designation. f_i = total number of failures of a given severity level in a given time interval, i =1,...
  Plot cumulative failures versus a suitable time base. The curve can be derived for the system as a whole, subsystems, or modules.

- Fault-Days Number: This measure represents the number of days that faults stays in a software system from their creation to their removal [45]:

  - "Phase when the fault was introduced in the system".
  - "Date when the fault was introduced in the system".
  - "Phase, date, and time when the fault is removed".

[5] This measure assumes that a structured design language is used. However, if some other design methodology is used, then some other unit of defect density has to be developed to conform to the methodology in which the design is expressed.

For measure the fault-days number we need to represents the number of days that faults spend in the software system from their creation to their removal with passing the following steps:

- Phase when the fault was introduced in the system.
- Date when the fault was introduced in the system.
- Phase, date, and time when the fault is removed [6].
- Fault-days: For each fault detected and removed, during any phase, the number of days from its creation to its removal is determined.
- The fault-days are then summed for all faults detected and removed, to get the fault-days number at system level, including all faults detected/removed up to the delivery date[7].
- The fault introduced during the requirements phase is assumed to have been created at the middle of the requirement phase because the exact time that the corresponding piece of requirement was specified is not known.
- The measure is calculated as follows: Fault days number $(FD) = \sum_i FD_i$.

5.1 Conclusion

Many different measures may apply to characteristics and sub-characteristics but, in most cases, they are not practical to collect automatically. We choose the measures that are the easiest to compute on programs architectures, related to design patterns.

We now have a design pattern-based quality model and metrics to measure its characteristics and sub-characteristics. We evaluate manually design patterns to link our quality model with concrete quality values.

[6] For more meaningful measures, time units can be relative to test or operational time.

[7] In cases if the creation date for the fault is not known, the fault is assumed to have been created at the middle of the phase in which it was introduced.

CHAPTER 6

DESIGN PATTERN EVALUATION

We summarize[1] the twenty-three design patterns and evaluate manually their quality characteristics using five-levels Lickert scale (*Excellent* , *Good*, *Fair*, *Bad* and *Very bad*). We use *N/A* for characteristics not applicable to some design patterns.

Our evaluation decomposes in two parts. First we introduce the evaluation for sub-characteristics such as Expandability, Simplicity, Generality, Modularity, Software-independence, Hardware-independence, Learnability, Understandability, and Operability. Then, we evaluate separately the Scalability and Robustness, because these characteristics are related to software operation and do not concern the architecture of program directly.

6.1 Creational Design Patterns

We evaluate now creational design patterns, providing "good" solutions to problems of creating objects.

6.1.1 Abstract Factory

- Intent: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- Applicability: Use the Abstract Factory pattern when:
  - A system should be independent of how its products are created, composed, and represented.
  - A system should be configured with one of multiple families of products.
  - A family of related product objects is designed to be used together, and you need to enforce this constraint.
  - You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.
- Structure:

[1] All subsequent definitions and pictures are from [24] and [38]

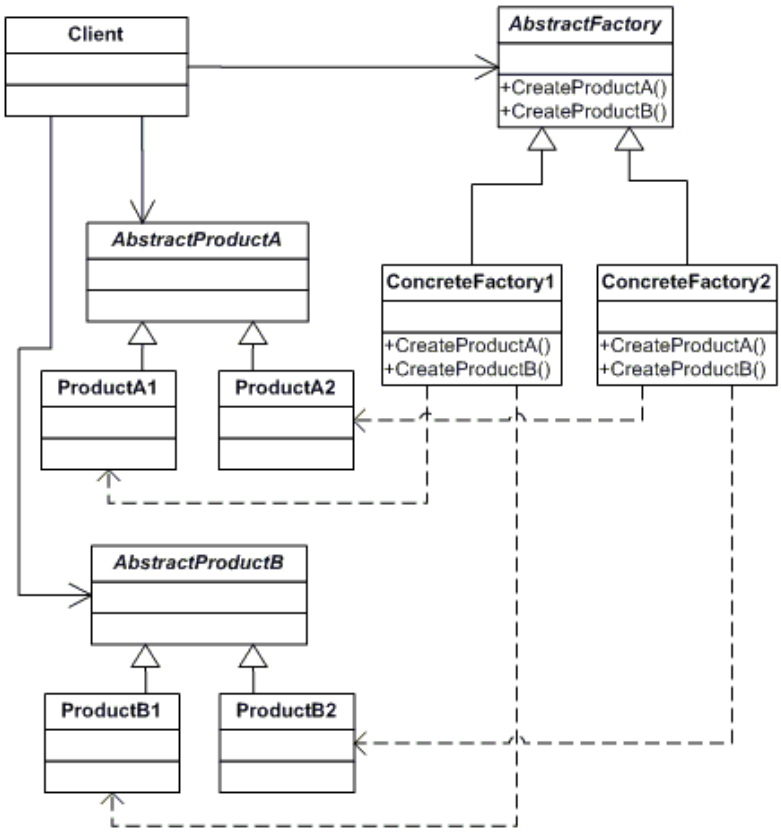

Figure 6.1: Abstract Factory UML-like class diagram

- Consequences: The Abstract Factory design pattern has the following benefits and liabilities:
  - It isolates concrete classes.
  - It makes exchanging product families easy.
  - It promotes consistency among products.
  - Supporting new kinds of products is difficult.
- Evaluation:

Quality sub-characteristics	Values
Expendability	*Excellent*
Simplicity	*Excellent*
Generality	*Good*
Modularity	*Good*
Software Independence	*Fair*
Hardware Independence	*Fair*
Learnability	*Good*
Understandability	*Good*
Operability	*Good*
Scalability	*Good*
Robustness	*Good*

6.1.2 Builder

- Intent: Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- Applicability: Use the Builder pattern when:
  - The algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled.
  - The construction process must allow different representations for the object that is constructed.
- Structure:

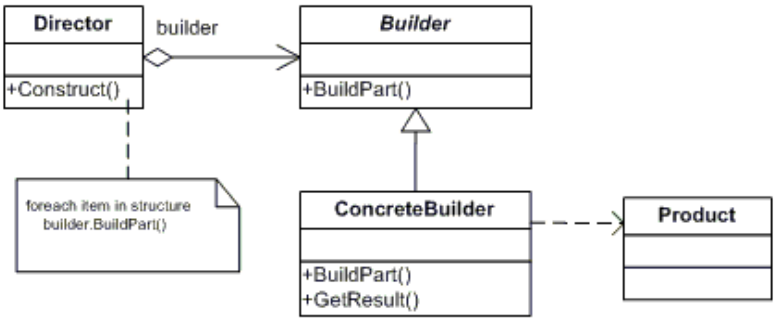

Figure 6.2: Builder design pattern UML-like class diagram

- Consequences:

  - It lets you vary a product's internal representation.
  - It isolates code for construction and representation.
  - It gives you finer control over the construction process.

- Evaluation

Quality sub-characteristics	Values
Expendability	*Good*
Simplicity	*Good*
Generality	*Fair*
Modularity	*Fair*
Software Independence	*N/A*
Hardware Independence	*N/A*
Learnability	*Fair*
Understandability	*Good*
Operability	*Fair*
Scalability	*Good*
Robustness	*Good*

6.1.3 Factory Method

- Intent: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Applicability: Use the Factory Method pattern when:
  - A class cannot anticipate the class of objects it must create.
  - A class wants its subclasses to specify the objects it creates.
  - Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.
- Structure:
- Consequences:
  - Provides hooks for subclasses.
  - Connects parallel class hierarchies.

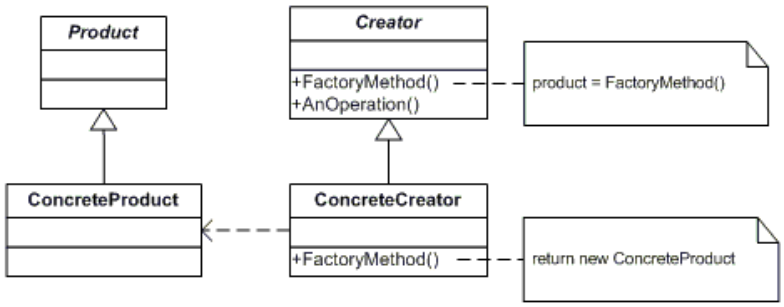

Figure 6.3: Factory Method design pattern UML-like class diagram

- Evaluation:

Quality sub-characteristics	Values
Expendability	*Bad*
Simplicity	*Bad*
Generality	*Fair*
Modularity	*Good*
Software Independence	*N/A*
Hardware Independence	*N/A*
Learnability	*Good*
Understandability	*Good*
Operability	*Good*
Scalability	*Good*
Robustness	*Good*

6.1.4 Prototype

- Intent: Specify the kind of objects to create using a prototypical instance, and create new objects by copying this prototype.
- Applicability: Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented; and:
  - When the classes to instantiate are specified at run-time, for example, by dynamic loading.
  - To avoid building a class hierarchy of factories that parallels the class hierarchy of products.
  - When instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of

prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

- Structure:

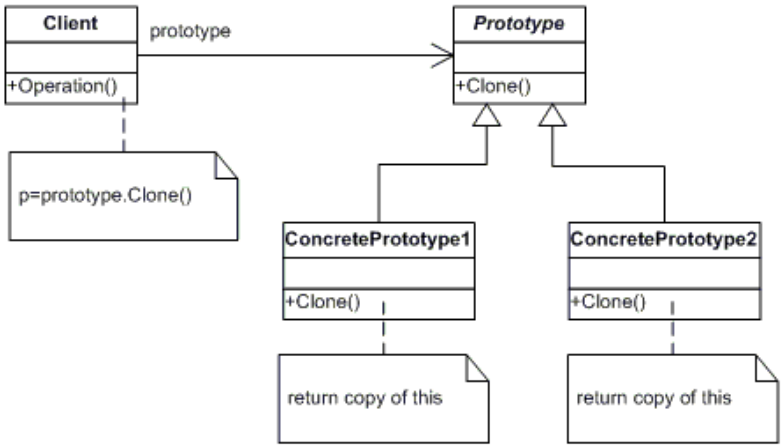

Figure 6.4: Prototype design pattern UML-like class diagram

- Consequences:
  - Adding and removing products at run-time.
  - Specifying new objects by varying values.
  - Specifying new objects by varying structure.
  - Reduced sub-classing.
  - Configuring an application with classes dynamically.
- Evaluation:

Quality sub-characteristics	Values
Expendability	*Excellent*
Simplicity	*Good*
Generality	*Fair*
Modularity	*Good*
Software Independence	*N/A*
Hardware Independence	*N/A*
Learnability	*Fair*
Understandability	*Good*
Operability	*Fair*
Scalability	*Excellent*
Robustness	*Good*

6.1.5 Singleton

- Intent: Ensure a class has only one instance and provide a global point of access to it.
- Applicability: Use the Singleton pattern when:
  - There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
  - When the sole instance should be extensible by sub-classing, and clients should be able to use an extended instance without modifying their code.
- Structure:

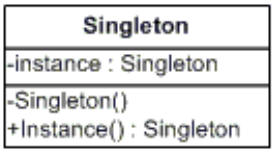

Figure 6.5: Singleton design pattern UML-like class diagram

- Consequences: The Singleton pattern has several benefits:
  - Controlled access to sole instance.
  - Reduced name space.
  - Permits refinement of operations and representation.

  - Permits a variable number of instances.
  - More flexible than class operations.

- Evaluation:

Quality sub-characteristics	Values
Expendability	*Bad*
Simplicity	*Very bad*
Generality	*Fair*
Modularity	*Excellent*
Software Independence	*Fair*
Hardware Independence	*Fair*
Learnability	*Fair*
Understandability	*Fair*
Operability	*Fair*
Scalability	*Good*
Robustness	*Good*

6.2 Structural Design Patterns

We evaluate the quality of design patterns concerned with the structure of programs.

6.2.1 Adapter

- Intent: Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

- Applicability: Use the Adapter pattern when

  - You want to use an existing class, and its interface does not match the one you need.
  - You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
  - You need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

- Structure:

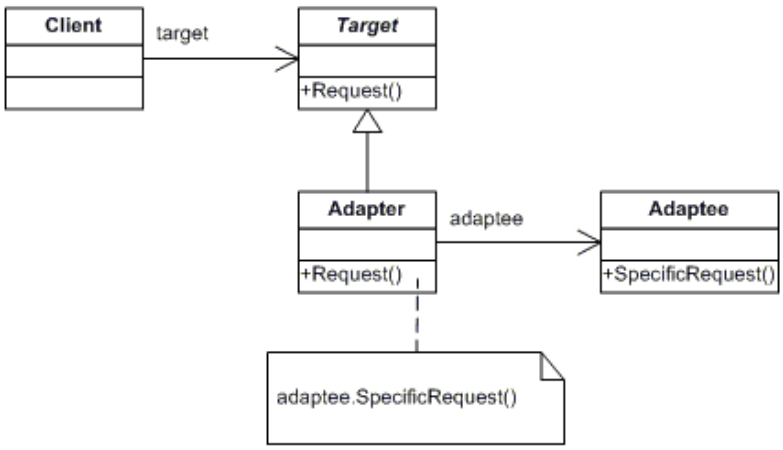

Figure 6.6: UML class diagram for Adapter pattern

- Consequences:
  - How much adapting does Adapter do?
  - Pluggable adapters.
  - Using two-way adapters to provide transparency

- Evaluation:

Quality sub-characteristics	Values
Expendability	*Fair*
Simplicity	*Fair*
Generality	*Bad*
Modularity	*Good*
Software Independence	*Good*
Hardware Independence	*N/A*
Learnability	*Good*
Understandability	*Fair*
Operability	*Fair*
Scalability	*Good*
Robustness	*Fair*

6.2.2 Bridge

- Intent: Decouple an abstraction from its implementation so that the two can vary independently.

- Applicability: Use the Bridge pattern when
  - You want to avoid a permanent binding between an abstraction and its implementation.
  - Both the abstractions and their implementations should be extensible by subclassing.
  - Changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled.
  - (C++) you want to hide the implementation of an abstraction completely from clients. In C++ the representation of a class is visible in the class interface.
  - You have a proliferation of classes as shown earlier in the first Motivation diagram. Such a class hierarchy indicates the need for splitting an object into two parts.
  - You want to share an implementation among multiple objects (perhaps using reference counting), and this fact should be hidden from the client.
- Structure:

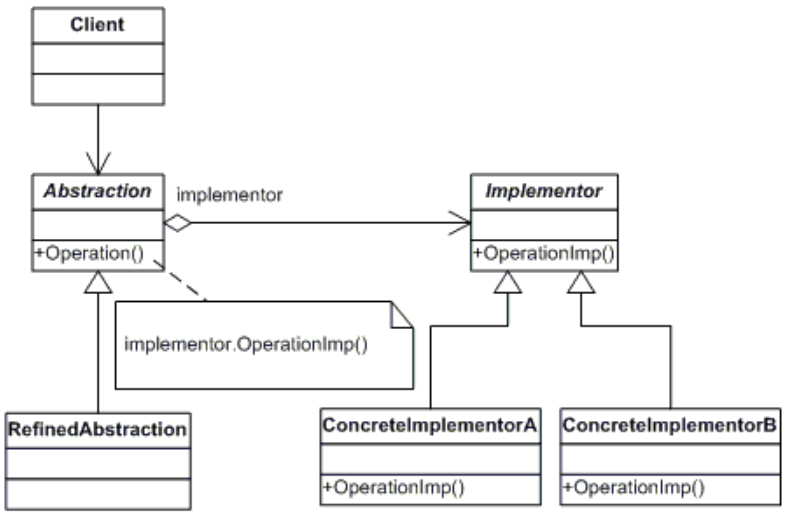

Figure 6.7: UML class diagram for Bridge pattern

- Consequences: The Bridge pattern has the following consequences:
  - Decoupling interface and implementation
  - Improved extensibility
  - Hiding implementation details from clients

- Evaluation:

Quality sub-characteristics	Values
Expendability	*Good*
Simplicity	*Fair*
Generality	*Good*
Modularity	*Good*
Software Independence	*N/A*
Hardware Independence	*N/A*
Learnability	*Fair*
Understandability	*Fair*
Operability	*Good*
Scalability	*Good*
Robustness	*Good*

6.2.3 Composite

- Intent: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Applicability: Use the Composite pattern when
  - You want to represent part-whole hierarchies of objects
  - You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.
- Structure:

  A typical Composite object structure might look like this:
- Consequences: The Composite pattern
  - Defines class hierarchies consisting of primitive objects and composite objects
  - Makes the client simple. Clients can treat composite structures and individual objects uniformly
  - Makes it easier to add new kinds of components
  - Can make your design overly general

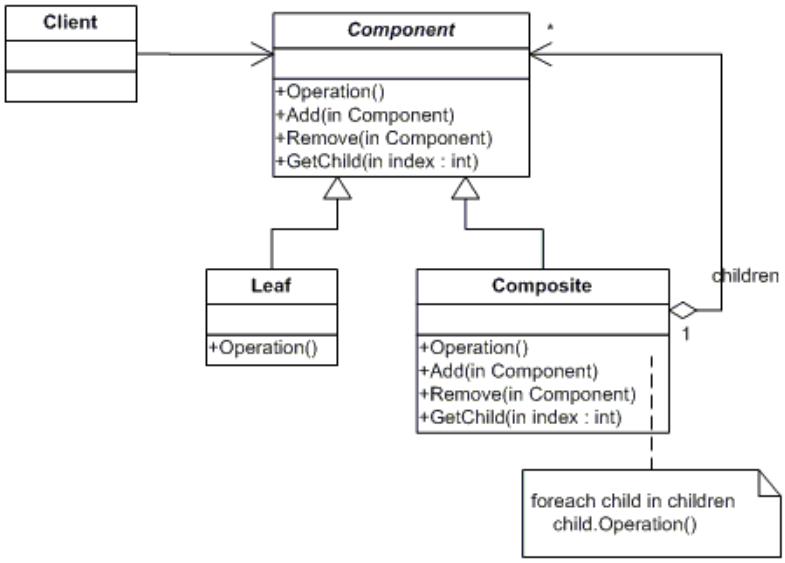

Figure 6.8: UML class diagram for Composite pattern

- Evaluation:

Quality sub-characteristics	Values
Expendability	*Fair*
Simplicity	*Fair*
Generality	*N/A*
Modularity	*Fair*
Software Independence	*N/A*
Hardware Independence	*N/A*
Learnability	*Fair*
Understandability	*Good*
Operability	*N/A*
Scalability	*N/A*
Robustness	*Good*

6.2.4 Decorator

- Intent: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Applicability: Use Decorator
  - To add responsibilities to individual objects dynamically and transparently,

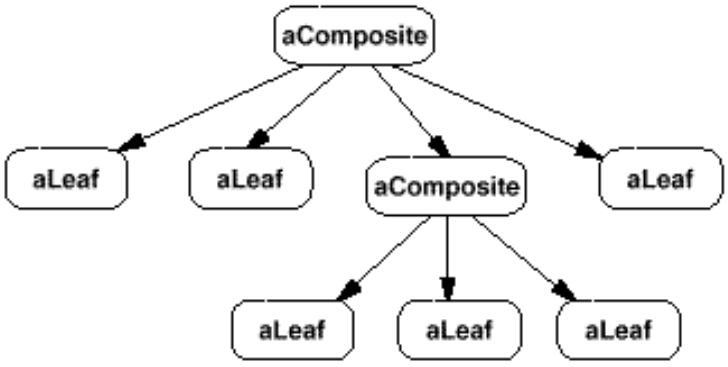

Figure 6.9: Typical Composite object structure

that is, without affecting other objects.

- For responsibilities that can be withdrawn.
- When extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for subclassing.

- Structure:
- Consequences: The Decorator pattern has at last two key benefits and two liabilities:
  - More flexibility than static inheritance
  - Avoids feature-laden classes high up in the hierarchy
  - A decorator and its component aren't identical
  - Lots of little objects
- Evaluation:

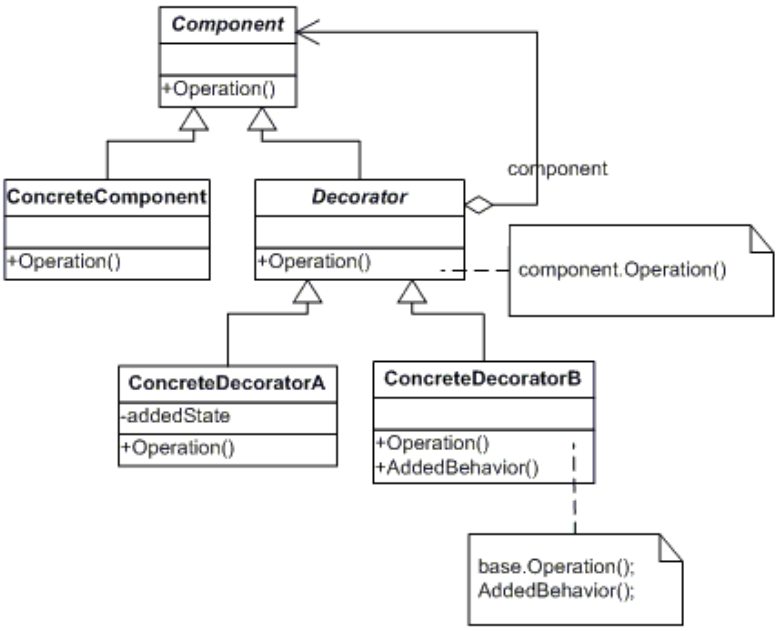

Figure 6.10: UML class diagram for Decorator pattern

Quality sub-characteristics	Values
Expendability	*Excellent*
Simplicity	*Excellent*
Generality	*Good*
Modularity	*Fair*
Software Independence	*Good*
Hardware Independence	*N/A*
Learnability	*Good*
Understandability	*Good*
Operability	*Good*
Scalability	*Good*
Robustness	*Fair*

6.2.5 Façade

- Intent: Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.
- Applicability: Use the Facade pattern when

  - You want to provide a simple interface to a complex subsystem.
  - There are many dependencies between clients and the implementation classes of an abstraction.
  - You want to layer your subsystems.

- Structure:

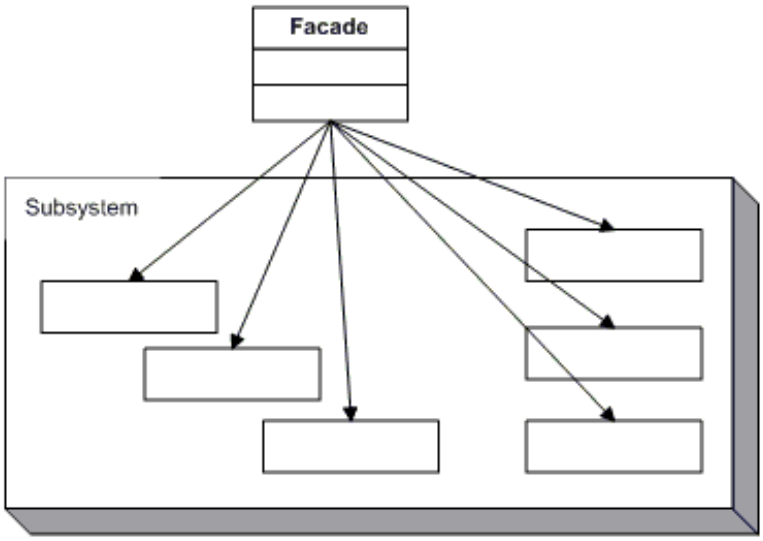

Figure 6.11: UML class diagram for Façade pattern

- Consequences: The Facade pattern offers the following benefits:
  - It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use
  - It promotes weak coupling between the subsystem and its clients
  - It doesn't prevent applications from using subsystem classes if they need to

- Evaluation:

Quality sub-characteristics	Values
Expendability	*Good*
Simplicity	*Good*
Generality	*Good*
Modularity	*Good*
Software Independence	*N/A*
Hardware Independence	*N/A*
Learnability	*Fair*
Understandability	*Good*
Operability	*Fair*
Scalability	*Fair*
Robustness	*Fair*

6.2.6 Flyweight

- Intent: Use sharing to support large numbers of fine-grained objects efficiently.
- Applicability: The Flyweight pattern's effectiveness depends heavily on how and where it's used. Apply the Flyweight pattern when all of the following are true:
  - An application uses a large number of objects
  - Storage costs are high because of the sheer quantity of objects
  - Most object state can be made extrinsic
  - Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed
  - The application doesn't depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects
- Structure: The following object diagram shows how flyweights are shared:
- Consequences: Flyweights may introduce run-time costs associated with transferring, finding, and/or computing extrinsic state, especially if it was formerly stored as intrinsic state. However, such costs are offset by space savings, which increase as more flyweights are shared. Storage savings are a function of several factors:
  - The reduction in the total number of instances that comes from sharing
  - The amount of intrinsic state per object
  - Whether extrinsic state is computed or stored

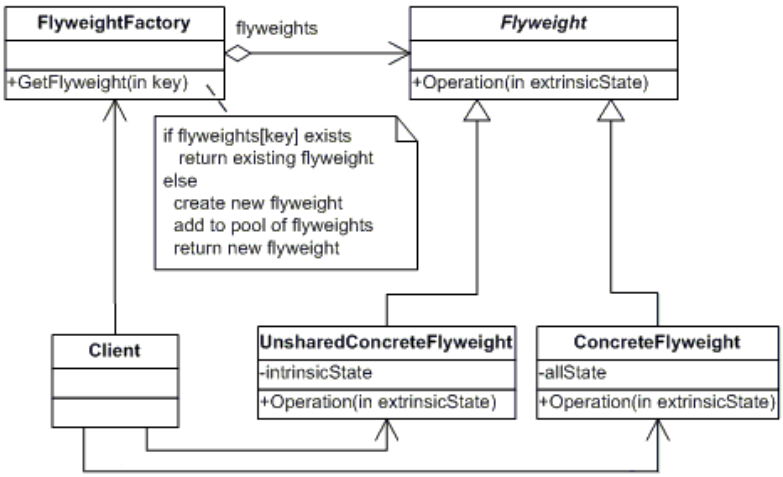

Figure 6.12: UML class diagram for Flyweight pattern

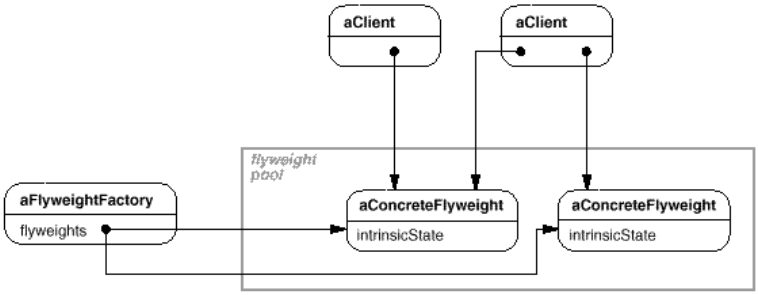

Figure 6.13: object diagram to shows the flyweights' share

- Evaluation:

Quality sub-characteristics	Values
Expendability	*Bad*
Simplicity	*Bad*
Generality	*Fair*
Modularity	*Good*
Software Independence	*N/A*
Hardware Independence	*N/A*
Learnability	*Good*
Understandability	*Bad*
Operability	*Fair*
Scalability	*Good*
Robustness	*Good*

6.2.7 Proxy

- Intent: Provide a surrogate or placeholder for another object to control access to it.
- Applicability: Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. Here are several common situations in which the Proxy pattern is applicable:
  - A remote proxy provides a local representative for an object in a different address space
  - A virtual proxy creates expensive objects on demand
  - A protection proxy controls access to the original object. Protection proxies are useful when objects should have different access rights
  - A smart reference is a replacement for a bare pointer that performs additional actions when an object is accessed
- Structure: Here's a possible object diagram of a proxy structure at run-time:

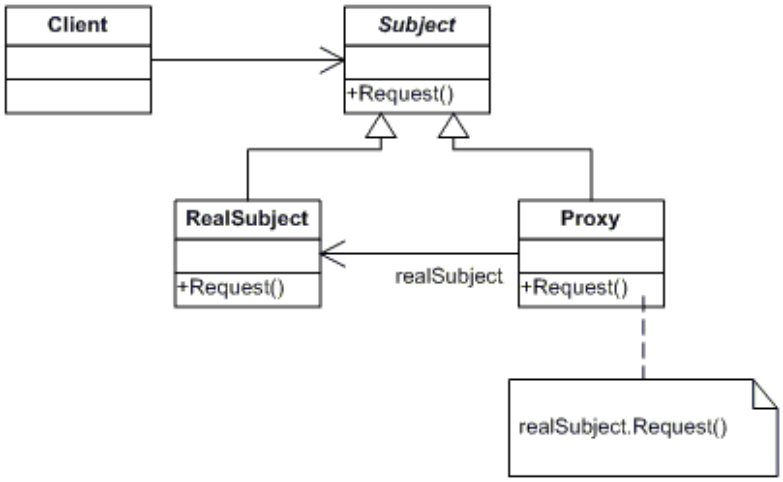

Figure 6.14: UML class diagram for Proxy pattern

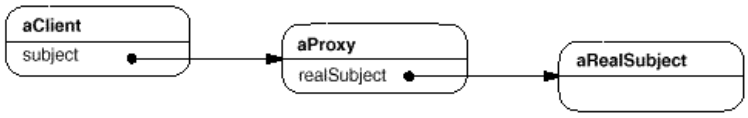

Figure 6.15: object diagram of a proxy structure at run-time

- Consequences: The Proxy pattern introduces a level of indirection when accessing an object. The additional indirection has many uses, depending on the kind of proxy:
  - A remote proxy can hide the fact that an object resides in a different address space
  - A virtual proxy can perform optimizations such as creating an object on demand
  - Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed
- Evaluation:

Quality sub-characteristics	Values
Expendability	*Good*
Simplicity	*Bad*
Generality	*Fair*
Modularity	*Good*
Software Independence	*N/A*
Hardware Independence	*N/A*
Learnability	*Fair*
Understandability	*Bad*
Operability	*Good*
Scalability	*Good*
Robustness	*Fair*

6.3 Behavioral Design Patterns

Behavioral describe patterns describe recurring behavioral object models.

6.3.1 Chain of Responsibility

- Intent: Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- Applicability: Use Chain of Responsibility when
  - More than one object may handle a request, and the handler isn't known a priori. The handler should be ascertained automatically

  - You want to issue a request to one of several objects without specifying the receiver explicitly
  - The set of objects that can handle a request should be specified dynamically

- Structure: A typical object structure might look like this:

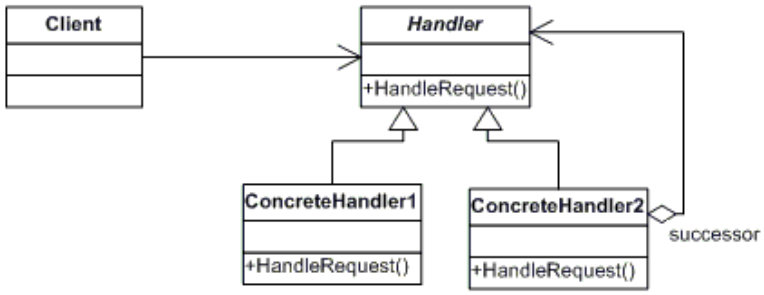

Figure 6.16: UML class diagram for Chain of Resp. pattern

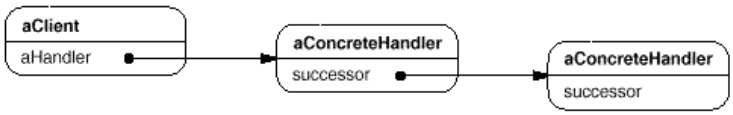

Figure 6.17: A typical object structure

- Consequences: Chain of Responsibility has the following benefits and liabilities:
  - Reduced coupling
  - Added flexibility in assigning responsibilities to objects
  - Receipt isn't guaranteed

- Evaluation:

Quality sub-characteristics	Values
Expendability	*Good*
Simplicity	*Good*
Generality	*Good*
Modularity	*Bad*
Software Independence	*N/A*
Hardware Independence	*N/A*
Learnability	*Fair*
Understandability	*Fair*
Operability	*Good*
Scalability	*Bad*
Robustness	*Fair*

6.3.2 Command

- Intent: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Applicability: Use the Command pattern when you want to
  - Parameterize objects by an action to perform, as MenuItem objects did above
  - Specify, queue, and execute requests at different times
  - Support undo. The Command's Execute operation can store state for reversing its effects in the command itself
  - Support logging changes so that they can be reapplied in case of a system crash
  - Structure a system around high-level operations built on primitives operations
- Structure:
- Consequences: The Command pattern has the following consequences:
  - Command decouples the object that invokes the operation from the one that knows how to perform it
  - Commands are first-class objects. They can be manipulated and extended like any other object
  - You can assemble commands into a composite command

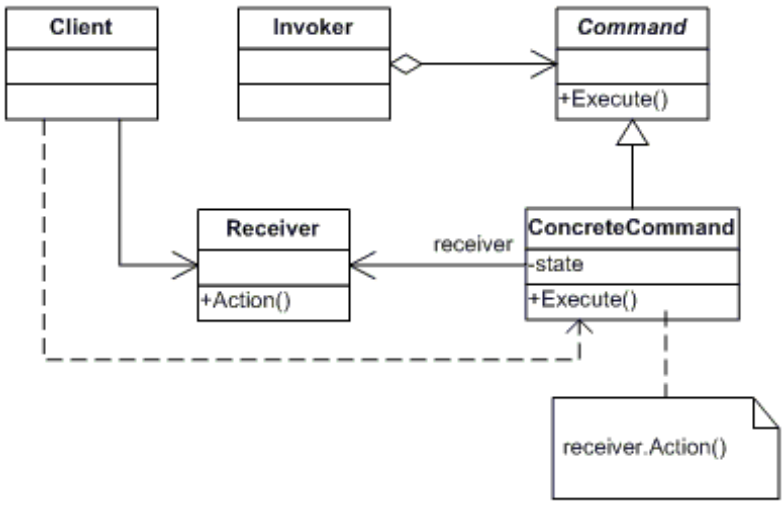

Figure 6.18: UML class diagram for Command pattern

- Evaluation:

Quality sub-characteristics	Values
Expendability	*Good*
Simplicity	*Bad*
Generality	*N/A*
Modularity	*N/A*
Software Independence	*N/A*
Hardware Independence	*N/A*
Learnability	*Bad*
Understandability	*Very bad*
Operability	*Good*
Scalability	*Good*
Robustness	*Good*

6.3.3 Interpreter

- Intent: Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- Applicability: Use the Interpreter pattern when there is a language to interpret, and you can represent statements in the language as abstract syntax trees. The Interpreter pattern works best when
  - The grammar is simple

  - Efficiency is not a critical concern

- Structure:

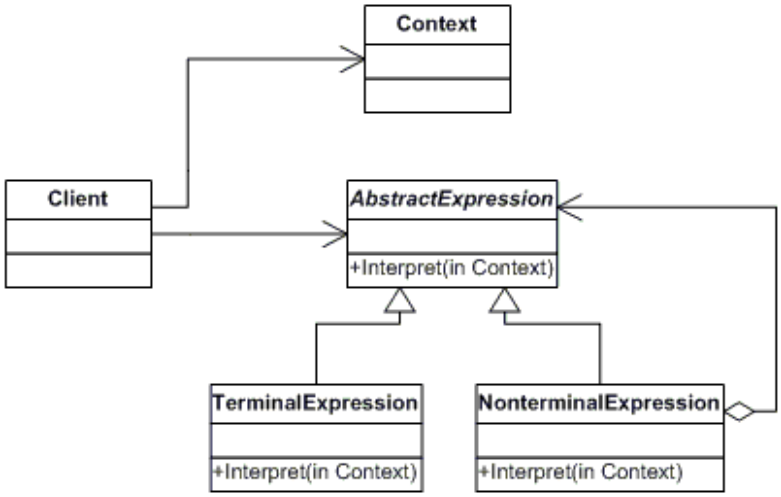

Figure 6.19: UML class diagram for Interpreter pattern

- Consequences: The Interpreter pattern has the following benefits and liabilities:
  - It's easy to change and extend the grammar
  - Implementing the grammar is easy, too
  - Complex grammars are hard to maintain
  - Adding new ways to interpret expressions
- Evaluation:

Quality sub-characteristics	Values
Expendability	*Good*
Simplicity	*Fair*
Generality	*Good*
Modularity	*Fair*
Software Independence	*N/A*
Hardware Independence	*N/A*
Learnability	*Fair*
Understandability	*Fair*
Operability	*Good*
Scalability	*Good*
Robustness	*Fair*

6.3.4 Iterator

- Intent: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Applicability: Use the Iterator pattern
  - To access an aggregate object's contents without exposing its internal representation.
  - To support multiple traversals of aggregate objects.
  - To provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).
- Structure:
- Consequences: The Iterator pattern has three important consequences:
  - It supports variations in the traversal of an aggregate
  - Iterators simplify the Aggregate interface
  - More than one traversal can be pending on an aggregate
- Evaluation:

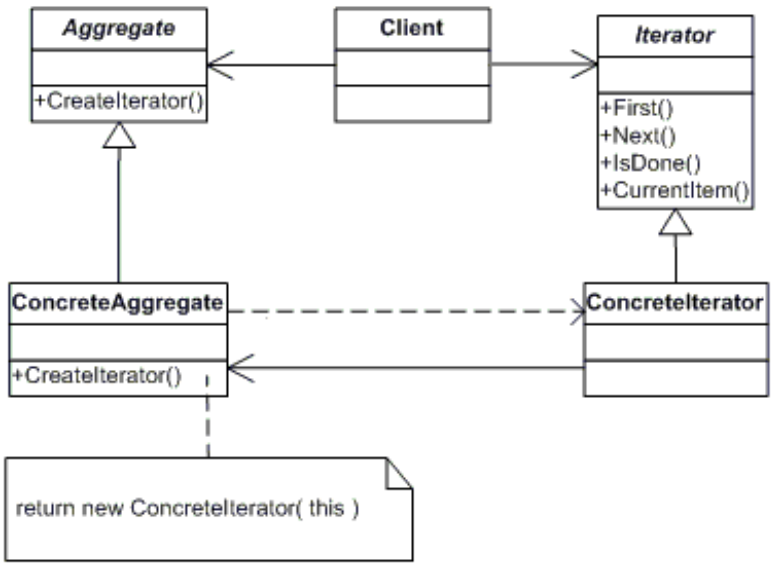

Figure 6.20: UML class diagram for Iterator pattern

Quality sub-characteristics	Values
Expendability	*Excellent*
Simplicity	*Excellent*
Generality	*Good*
Modularity	*Fair*
Software Independence	*Good*
Hardware Independence	*N/A*
Learnability	*Good*
Understandability	*Fair*
Operability	*Fair*
Scalability	*Good*
Robustness	*Good*

6.3.5 Mediator

- Intent: Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- Applicability: Use the Mediator pattern when
  - A set of objects communicate in well-defined but complex ways

  - Reusing an object is difficult because it refers to and communicates with many other objects
  - A behavior that's distributed between several classes should be customizable without a lot of subclassing

- Structure: A typical object structure might look like this:

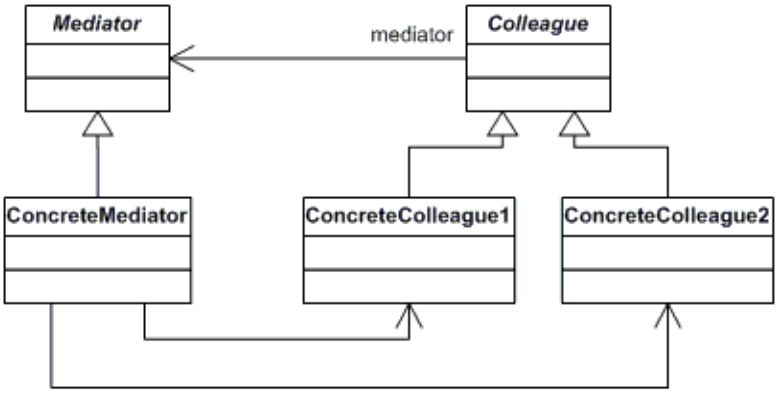

Figure 6.21: UML class diagram for Mediator pattern

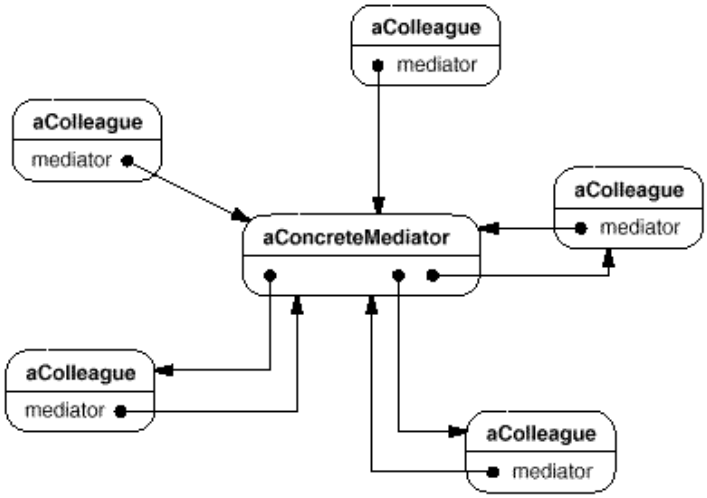

Figure 6.22: A typical object structure

- Consequences: The Mediator pattern has the following benefits and drawbacks:
  - It limits subclassing

  - It decouples colleagues
  - It simplifies object protocols
  - It abstracts how objects cooperate
  - It centralizes control

- Evaluation:

Quality sub-characteristics	Values
Expendability	*Good*
Simplicity	*Fair*
Generality	*Good*
Modularity	*Good*
Software Independence	*N/A*
Hardware Independence	*N/A*
Learnability	*Fair*
Understandability	*Fair*
Operability	*Good*
Scalability	*Good*
Robustness	*Fair*

6.3.6 Memento

- Intent: Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

- Applicability: Use the Memento pattern when

  - A snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, and
  - A direct interface to obtaining the state would expose implementation details and break the object's encapsulation

- Structure:

- Consequences: The Memento pattern has several consequences:

  - Preserving encapsulation boundaries
  - It simplifies Originator
  - Using mementos might be expensive

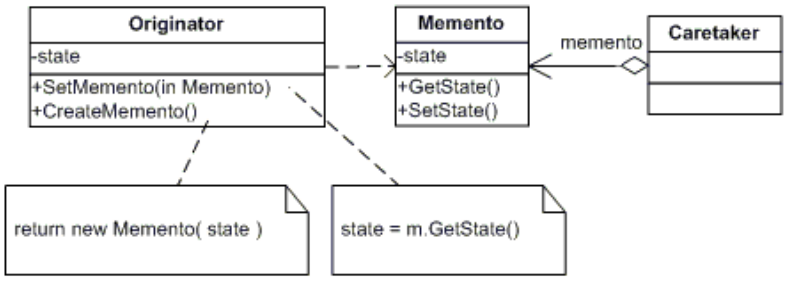

Figure 6.23: UML class diagram for Memento pattern

  - Defining narrow and wide interfaces
  - Hidden costs in caring for mementos

- Evaluation:

Quality sub-characteristics	Values
Expendability	*Good*
Simplicity	*Fair*
Generality	*Fair*
Modularity	*Very bad*
Software Independence	*N/A*
Hardware Independence	*N/A*
Learnability	*Bad*
Understandability	*Fair*
Operability	*Good*
Scalability	*Fair*
Robustness	*Bad*

6.3.7 Observer

- Intent: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

- Applicability: Use the Observer pattern in any of the following situations:

  - When an abstraction has two aspects, one dependent on the other
  - When a change to one object requires changing others, and you don't know how many objects need to be changed
  - When an object should be able to notify other objects without making assumptions about who these objects are

- Structure:

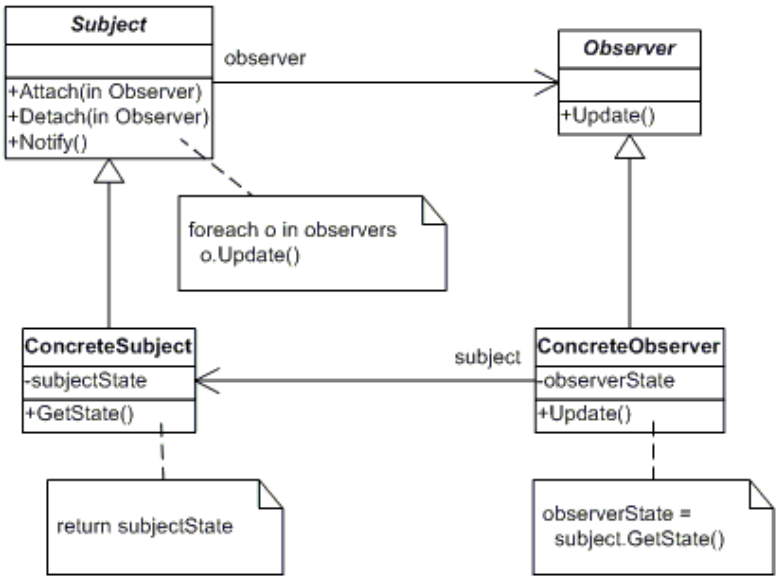

Figure 6.24: UML class diagram for Observer pattern

- Consequences: Further benefits and liabilities of the Observer pattern include the following:
  - Abstract coupling between Subject and Observer
  - Support for broadcast communication
  - Unexpected updates
- Evaluation:

Quality sub-characteristics	Values
Expendability	*Excellent*
Simplicity	*Good*
Generality	*Excellent*
Modularity	*N/A*
Software Independence	*N/A*
Hardware Independence	*N/A*
Learnability	*Fair*
Understandability	*Good*
Operability	*Good*
Scalability	*Good*
Robustness	*Good*

6.3.8 State

- Intent: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Applicability: Use the State pattern in either of the following cases:
  - An object's behavior depends on its state, and it must change its behavior at run-time depending on that state
  - Operations have large, multiparty conditional statements that depend on the object's state
  - Often, several operations will contain this same conditional structure
- Structure:

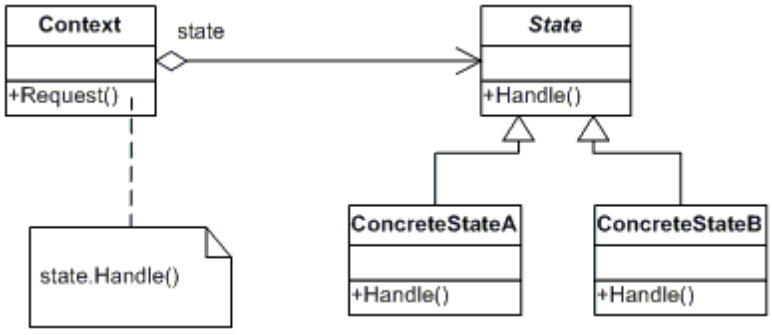

Figure 6.25: UML class diagram for State pattern

- Consequences: The State pattern has the following consequences:
  - It localizes state-specific behavior and partitions behavior for different states
  - It makes state transitions explicit
  - State objects can be shared
- Evaluation:

Quality sub-characteristics	Values
Expendability	*Good*
Simplicity	*Good*
Generality	*Fair*
Modularity	*Bad*
Software Independence	*Good*
Hardware Independence	*N/A*
Learnability	*Fair*
Understandability	*Very bad*
Operability	*Good*
Scalability	*Good*
Robustness	*Fair*

6.3.9 Strategy

- Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Applicability: Use the Strategy pattern when
  - Many related classes differ only in their behavior
  - You need different variants of an algorithm
  - An algorithm uses data that clients shouldn't know about
  - A class defines many behaviors, and these appear as multiple conditional statements in its operations
- Structure:
- Consequences: The Strategy pattern has the following benefits and drawbacks:
  - Families of related algorithms

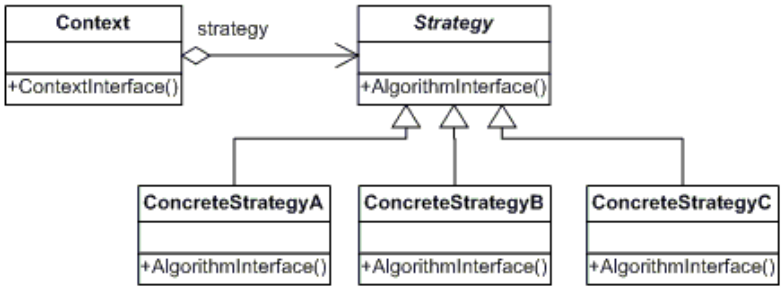

Figure 6.26: UML class diagram for Strategy pattern

  - An alternative to subclassing
  - Strategies eliminate conditional statements
  - A choice of implementations
  - Clients must be aware of different Strategies
  - Communication overhead between Strategy and Context
  - Increased number of objects

- Evaluation:

Quality sub-characteristics	Values
Expendability	*Good*
Simplicity	*Fair*
Generality	*Bad*
Modularity	*Fair*
Software Independence	*Fair*
Hardware Independence	*N/A*
Learnability	*Bad*
Understandability	*Bad*
Operability	*Fair*
Scalability	*Bad*
Robustness	*Fair*

6.3.10 Template Method

- Intent: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

- Applicability: The Template Method pattern should be used

  - To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary
  - When common behavior among subclasses should be factored and localized in a common class to avoid code duplication
  - To control subclasses extensions

- Structure:

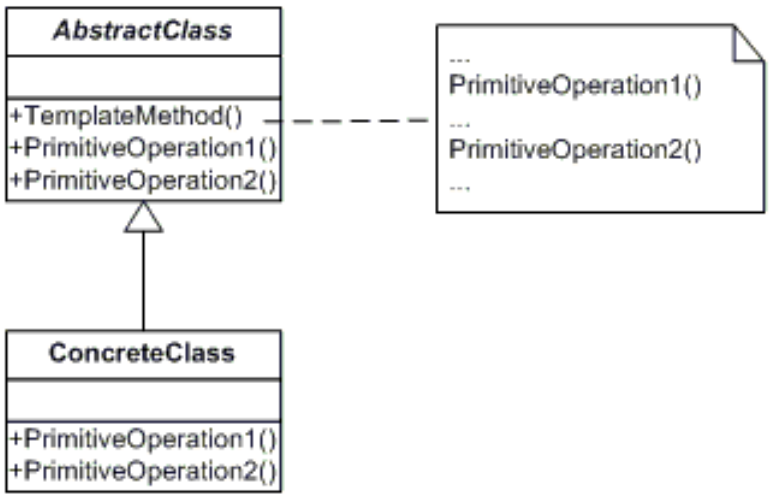

Figure 6.27: UML class diagram for Template Method pattern

- Consequences: Template methods call the following kinds of operations:
  - Concrete operations (either on the ConcreteClass or on client classes);
  - Concrete AbstractClass operations (*i.e.,* operations that are generally useful to subclasses);
  - Primitive operations (*i.e.,* abstract operations);
  - Factory methods, see Factory Method; and
  - Hook operations, which provide default behavior that subclasses can extend if necessary. A hook operation often does nothing by default

- Evaluation:

Quality sub-characteristics	Values
Expendability	*Excellent*
Simplicity	*Good*
Generality	*Fair*
Modularity	*N/A*
Software Independence	*N/A*
Hardware Independence	*N/A*
Learnability	*Good*
Understandability	*Good*
Operability	*Good*
Scalability	*Good*
Robustness	*Good*

6.3.11 Visitor

- Intent: Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- Applicability: Use the Visitor pattern when
  - An object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes
  - Many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations
  - The classes defining the object structure rarely change, but you often want to define new operations over the structure
- Structure:
- Consequences: Some of the benefits and liabilities of the Visitor pattern are as follows:
  - Visitor makes adding new operations easy
  - A visitor gathers related operations and separates unrelated ones
  - Adding new ConcreteElement classes is hard

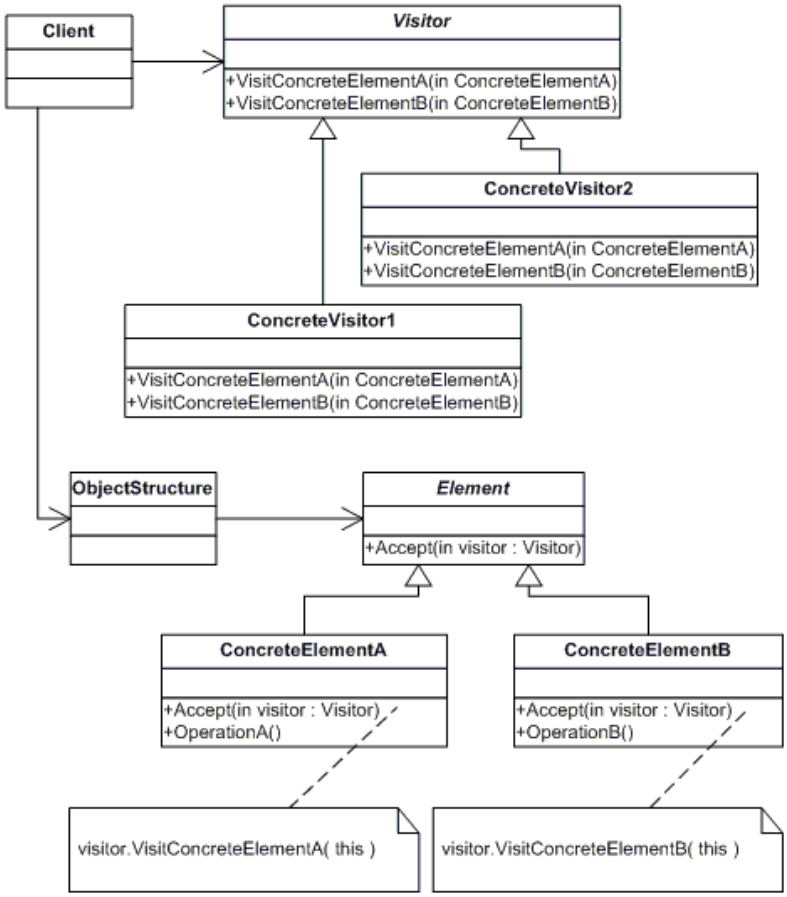

Figure 6.28: UML class diagram for Visitor pattern

  - Visiting across class hierarchies
  - Accumulating state
  - Breaking encapsulation

- Evaluation:

Quality sub-characteristics	Values
Expendability	*Excellent*
Simplicity	*Good*
Generality	*Good*
Modularity	*Fair*
Software Independence	*Good*
Hardware Independence	*N/A*
Learnability	*Good*
Understandability	*Bad*
Operability	*Fair*
Scalability	*Good*
Robustness	*Fair*

6.4 Conclusion

We believe that the problem of quality with design patterns comes both from the design patterns themselves and from their misuse. Unfortunately, little work has attempted so far to study the quality characteristics of design patterns rigorously.

We perform a thorough and systematic study of the 23 design patterns from the Gang of Four to assess the resulting quality characteristics. Table 6.1 is summarizes the evaluation. We found that the design patterns have different impact on quality.

This task is mandatory to build a quality model using patterns to consider program architectures. However, it is a difficult task and a task subject to the subjectivity of the analyst. We took pain to avoid bias but we can only argue that this study is the first thorough study of the quality design patterns, not the ultimate such study. We hope to see in the future other such studies to compare our analyses with others'.

	Quality sub-characteristics									Quality characteristics	
Design Patterns	Expendability	Simplicity	Generality	Modularity	Software Independence	Hardware Independence	Learnability	Understandability	Operability	Scalability	Robustness
Abstract Factory	*Excellent*	*Excellent*	*Good*	*Good*	*Fair*	*Fair*	*Good*	*Good*	*Good*	*Good*	*Good*
Builder	*Good*	*Good*	*Fair*	*Fair*	*N/A*	*N/A*	*Fair*	*Good*	*Fair*	*Good*	*Good*
Factory Method	*Bad*	*Bad*	*Fair*	*Good*	*N/A*	*N/A*	*Good*	*Good*	*Good*	*Good*	*Good*
Prototype	*Excellent*	*Good*	*Fair*	*Good*	*N/A*	*N/A*	*Fair*	*Good*	*Fair*	*Excellent*	*Good*
Singleton	*Bad*	*Very bad*	*Fair*	*Excellent*	*Fair*	*Fair*	*Fair*	*Fair*	*Fair*	*Good*	*Good*
Adapter	*Fair*	*Fair*	*Bad*	*Good*	*Good*	*N/A*	*Good*	*Fair*	*Fair*	*Good*	*Fair*
Bridge	*Good*	*Fair*	*Good*	*Good*	*N/A*	*N/A*	*Fair*	*Fair*	*Good*	*Good*	*Good*
Composite	*Fair*	*Fair*	*N/A*	*Fair*	*N/A*	*N/A*	*Fair*	*Good*	*N/A*	*N/A*	*Good*
Decotator	*Excellent*	*Excellent*	*Good*	*Fair*	*Good*	*N/A*	*Good*	*Good*	*Good*	*Good*	*Fair*
Façade	*Good*	*Good*	*Good*	*Good*	*N/A*	*N/A*	*Fair*	*Good*	*Fair*	*Fair*	*Fair*
Flyweight	*Bad*	*Bad*	*Fair*	*Good*	*N/A*	*N/A*	*Good*	*Bad*	*Fair*	*Good*	*Good*
Proxy	*Good*	*Bad*	*Fair*	*Good*	*N/A*	*N/A*	*Fair*	*Bad*	*Good*	*Good*	*Fair*
Chain of Responsibility	*Good*	*Good*	*Good*	*Bad*	*N/A*	*N/A*	*Fair*	*Fair*	*Good*	*Bad*	*Fair*
Command	*Good*	*Bad*	*N/A*	*N/A*	*N/A*	*N/A*	*Bad*	*Very bad*	*Good*	*Good*	*Good*
Interpreter	*Good*	*Fair*	*Good*	*Fair*	*N/A*	*N/A*	*Fair*	*Fair*	*Good*	*Good*	*Fair*
Iterator	*Excellent*	*Excellent*	*Good*	*Fair*	*Good*	*N/A*	*Good*	*Fair*	*Fair*	*Good*	*Good*
Mediator	*Good*	*Fair*	*Good*	*Good*	*N/A*	*N/A*	*Fair*	*Fair*	*Good*	*Good*	*Fair*
Memento	*Good*	*Fair*	*Fair*	*Very bad*	*N/A*	*N/A*	*Bad*	*Fair*	*Good*	*Fair*	*Bad*
Observer	*Excellent*	*Good*	*Excellent*	*N/A*	*N/A*	*N/A*	*Fair*	*Good*	*Good*	*Good*	*Good*
State	*Good*	*Good*	*Fair*	*Bad*	*Good*	*N/A*	*Fair*	*Very bad*	*Good*	*Good*	*Fair*
Strategy	*Good*	*Fair*	*Bad*	*Fair*	*Fair*	*N/A*	*Bad*	*Bad*	*Fair*	*Bad*	*Fair*
Template Method	*Excellent*	*Good*	*Fair*	*N/A*	*N/A*	*N/A*	*Good*	*Good*	*Good*	*Good*	*Good*
Visitor	*Excellent*	*Good*	*Good*	*Fair*	*Good*	*N/A*	*Good*	*Bad*	*Fair*	*Good*	*Fair*

Table 6.1: Design patterns quality characteristics evaluation

CHAPTER 7

QUALITY MODEL CONSIDERING PROGRAM ARCHITECTURES

It is betting that it is better to do a simple thing today
and pay a little more tomorrow
to change it if it needs it,
than to do a more complicated thing today
that may never be used anyway
Peter Wendorff.

7.1 Our Approach

Existing quality models attempt to link internal attributes of classes and external quality characteristics with little regard for the actual architectures of the programs. Thus, these quality models can distinguish hardly between well-structured programs and programs with poor architectures.

If we were in art rather than in informatics, we would say that existing quality models use identical quality models to compare a cubist painting, such as "Femme Profile" by Pablo Picasso (1939), with a realist picture, as shown in Figure 7.1. The two faces possess two eyes, one nose, two ears, and one mouth but with very different organisations, none being more beautiful, only more beautiful in different contexts.

Figure 7.1: A woman's profile: Cubist (left) and realist versions (right)

We want to build a quality model that considers quality characteristics covering software maintenance. Thus, we chose external quality characteristics related to maintenance and software metrics to fill the space between characteristics and software artifacts. We use design patterns as a basis to choose quality characteristics.

7.1.1 Underlying Assumptions

Underlying assumptions represent the choices made while building the quality model.

7.1.1.1 Human Factor

Some existing quality models can predict fault-proneness with reasonable accuracy in certain contexts. Other quality models attempt at evaluating several quality characteristics but fail at providing reasonable accuracy, from lack of data mainly.

We believe that quality models must evaluate high-level quality characteristics with great accuracy in terms well-known to software engineers to help maintainers in assessing programs and thus in predicting maintenance effort.

Such quality models can also help developers in building better quality programs by exposing the relationships between internal attributes and external quality characteristics clearly.

We take a less "quantitative" approach than quality models counting, for example, numbers of errors per classes and linking these numbers with internal attributes. We favour a more "qualitative" approach linking quality characteristics related to the maintainers' perception and work directly.

7.1.1.2 Quality Theory

Unfortunately, software engineering is well-known for its lack of theories. Software engineers do not have theories to support their work on development, maintenance, and to explain quality yet.

Thus, it is important to gather as much help as possible. Patterns, design patterns especially, are an interesting bridge between internal attributes of programs, external quality characteristics, and software engineers.

Indeed, when considering the lack of theories, patterns are an interesting tool to link internal attributes (concrete implementation of programs) in the one hand and subjective quality characteristics (subjective perceptions on programs) on the other hand.

7.1.1.3 Program Architecture

Pairwise dependencies among classes and internal attributes of classes are not enough: The organisations of classes, the program architectures, are important because they are the first things software engineers see and deal with.

A large body of work exist on program architecture, in particular on architectural drift or decay [88], which aims at analysing, organising, and tracking the modifications that architectures must undergo to keep them easy to understand and to modify, and thus to reduce maintenance effort [53].

However, to our best knowledge, no work attempted to develop quality models using programs internal attributes while considering their architectures explicitly. We try to

build such a quality model using patterns because patterns are well-known to improve program architectures and because software engineers use patterns, even unconsciously, when developing and maintaining programs [38, page xiii].

7.1.2 Process of Building a Quality Model

The process of building a quality model decomposes in three main tasks generally:

- Choosing and organising characteristics related to software maintenance.
- Choosing internal attributes that are computable with metrics.
- Linking quality characteristics with internal attributes to produce evaluation rules.

The process of building a quality model decomposes in the following tasks when using patterns to consider program architectures:

1. *Identifying the quality characteristics shared by a set of patterns which make programs more maintainable.* This task consists in identifying quality characteristics and sub-characteristics related to some patterns of interest. Among all possible characteristics, we can focus on characteristics for program maintenance.

2. *Organising the quality characteristics identified from the patterns.* This task consists in organising quality characteristics and sub-characteristics hierarchically [30] to build a quality model which can be linked with software artifacts using metrics.

3. *Choosing internal attributes relevant to patterns and their quality characteristics.* This task consists in choosing internal attributes which can be measured with metrics. The internal attributes must relate to the quality model from task 2, to link software artifacts with quality characteristics.

4. *Identifying programs implementing the patterns.* This task consists in identifying a set of programs in which developers used the patterns of interest. We name this set of base programs $\mathcal{BP}$.

5. *Assessing the quality of patterns using the quality model built in task 2.* This task consists in assessing the quality of patterns in the set of base programs $\mathcal{BP}$ manually, with the characteristics and sub-characteristics of the quality model.

6. *Computing the metrics identified in task 3 on the patterns in the programs identified in task 5.* This task consists in computing metric values for the patterns of interest identified in $\mathcal{BP}$. If class-based metrics are used, then we can compute the metric

values of the patterns as the average or as the variance of the class-based metric values.

7. *Linking the metric values computed in task 6 and the evaluation of the quality sub-characteristics and characteristics performed in task 5.* This task consists in applying a machine learning technique to link internal attributes of programs measured with metric values computed in task 6 and the evaluation of the quality sub-characteristics and characteristics from task 5.

8. *Validating the obtained quality model on other well-known instances of patterns.* This task consists in applying the evaluation rules from task 7 on other well-known programs to assess the evaluative power of the quality model. Using patterns, we must apply our quality model on well-known uses of patterns in programs with known quality characteristics.

The results of the eight previous tasks is a quality model which can evaluate the quality characteristics of programs while considering the program architectures through the evaluation of the quality of patterns. The quality model decomposes in several rules for each quality sub-characteristics and characteristics. These rules depends on different metrics to assess quality.

Figure 7.2 displays a simplified version of our process. First, we identify programs implementing patterns. Second, we identify in these programs the patterns used. Third, we evaluate the quality sub-characteristics and characteristics of the patterns manually. Fourth, we compute metrics for each identified patterns (by averaging class-based metrics, for example). Fifth, we use machine learning techniques to link metric values with the quality sub-characteristics and characteristics of patterns.

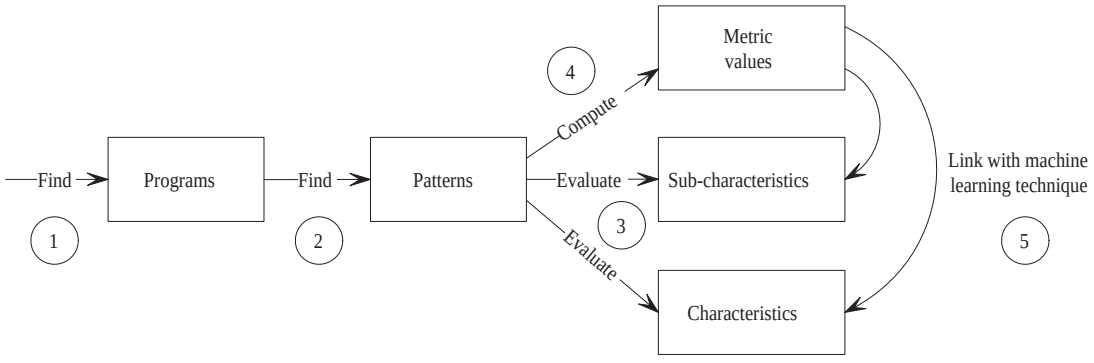

Figure 7.2: Simplified process of building a quality model considering program architectures

7.1.3 Process of Applying a Quality Model

We built a quality model using patterns to consider the quality of program architectures in addition to internal attributes of classes or couple thereof.

Applying such a quality model requires to consider subsets of a program architecture—micro-architectures—and to apply the quality model on these micro-architectures to assess their quality.

Again, if we were in art, we could say that existing quality models assess the quality characteristics of paintings by looking at many tiny parts of the painting (classes in program architectures) rather than by looking at larger pieces of the painting (patterns in program architectures), such as sketched in Figure 7.3.

Figure 7.3: Level of details considered in existing software quality model (left) versus in quality models based on patterns (right)

Thus, applying our quality model requires the four following tasks:

1. *Identifying micro-architectures similar to patterns in the architecture of the program $\mathcal{P}$ under evaluation.* There are many techniques existing to identify patterns in programs, for examples logic programming [93] or constraint programming [41].

2. *Measuring the internal attributes of each classes of the micro-architectures with metrics and averaging class-based metric values, if needed.* This task is straightforward, many tools existing to apply metrics on programs.

3. *Adapting the rules built from $\mathcal{BP}$ to $\mathcal{P}$ by computing the ratio between the metric values from $\mathcal{BP}$ and the metric values from $\mathcal{P}$.* This task consists in adapting the rules associated with the quality model built from $\mathcal{BP}$. Indeed, the rules are built from metric values with a certain minimum and maximum values depending on $\mathcal{BP}$, these values differ from the minimum and maximum values for $\mathcal{P}$. We compute the ratio between $min_{\mathcal{BP}}$ and $max_{\mathcal{BP}}$, on the one hand, and $min_{\mathcal{P}}$ and $max_{\mathcal{P}}$, on the other hand. Figure 7.4 illustrates rule adaptation. Yet again, if we were in art and we would like to compare the eyes in two different paintings, we would adapt the scales of the eyes before making any comparison.

4. *Applying our quality model on the identified micro-architectures.* This task consists in applying the rules adapted from the quality model on the metric values computed for the micro-architectures found in program $\mathcal{P}$.

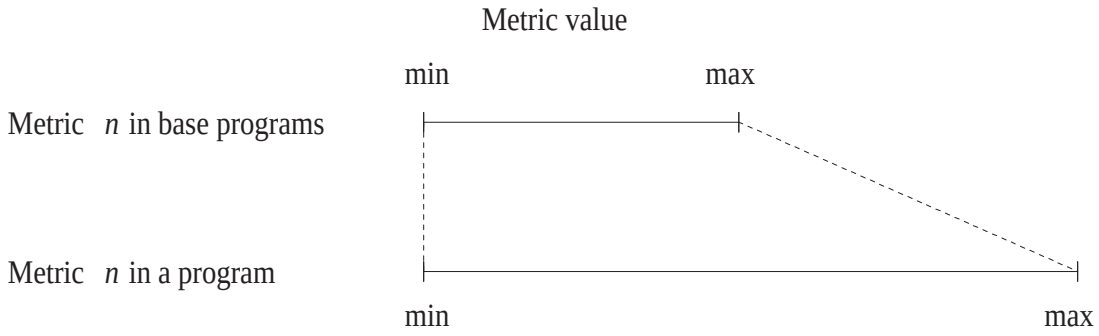

Figure 7.4: Adapting the rules of the quality model, ratio between minimum and maximum metric values of $\mathcal{BP}$ and $\mathcal{P}$

7.1.4 Discussion

The use of patterns brings an extra level of abstraction to the building of our quality model with respect to existing quality models. Indeed, we use patterns for three purposes: First, we survey quality characteristics of patterns theoretically to define and to organise the quality characteristics of our quality model; Second, we validate our quality model on well-known instances of patterns; Third, we apply our quality model on micro-architectures similar to patterns.

We can use our quality model on any micro-architecture, independently of the micro-architectures sizes and organisations, and thus potentially on complete program architectures. This use is similar to the use of existing quality models which are built using internal attributes and quality characteristics of given programs but applied on similar yet sometimes very differing programs.

The use of patterns as a basis to build a quality model results in our choice to study qualitative quality characteristics over quantitative characteristics, such as fault-proneness. Thus, we want to build a quality model tailored for maintainers, evaluating "qualitative" characteristics with which maintainers can predict maintenance effort.

We choose patterns because developers always use patterns. Indeed, developers always use recurring solutions to solve design problems. Thus, patterns are an integral part of any reasonably well-developed programs [38].

However, the use of pattern is but a step towards quality models that can evaluate software quality while considering the architectures of programs. Indeed, a quality model built using patterns assesses the quality of programs through larger *parts* than existing quality models because it uses patterns instead of classes. Yet, it does not consider the overall architectures of the programs: It is similar in art to assessing the quality of a painting using parts rather than looking at the whole picture, such as in the right-hand side of Figure 7.3.

The identification of micro-architectures similar to patterns is the subject of many work. It is often a complex and resource-consuming activity. However, we can also apply our quality model on any micro-architectures from a program architecture to evaluate their quality characteristics. Indeed, the quality model is built considering patterns to assess quality characteristics of *any* micro-architecture in a program.

7.2 Piecemeal Quality Assessment

The following steps summarise the concrete building of rules related to our quality model considering programs architectures using design patterns. Figure 7.5 presents general information and a synthetic view on our software quality evaluation method.

7.2.1 Dependent Variables

The dependent variables in our quality evaluation are:

- The quality characteristics related to a super-characteristics.
- The group of people which quality will be measuring for them.

7.2.2 Independent Variables

The independent variables in our quality model are the internal attributes which can be measured by software metrics. These internal attributes are similar to those in other quality models from the literature: size, filiation, cohesion, coupling, and complexity.

7.2.3 Analysis Technique

We use a propositional rule learner algorithm, JRip. JRip is Weka—an open-source program collecting machine learning algorithms for data mining tasks [91]—implementation of the Ripper rule learner. It is a fast algorithm for learning "If–Then" rules. Like decision trees, rule learning algorithms are popular because the knowledge representation is easy to interpret.

7.2.4 Step 1: Create a Measurement Team

The measurement team will consist of a small group (at least a person) who are committed to represent the category of people for which our software evaluation will be implement. In our case, we consider software developers and maintainers.

7.2.5 Step 2: Identifying Sample Programs

We must choose a simple programs ($\mathcal{BP}$) to be considered as sample evaluation set of our model.

7.2.6 Step 3: Building a Quality Model

The process of building a quality model decomposes in two main tasks generally:

- Choosing a super-characteristic.
- Choosing and organising characteristics related to super-characteristic.

In our case study, we consider design patterns especially as bridge between internal attributes of programs, external quality characteristics, and software engineers.

7.2.7 Step 4: Human Evaluation

The small group, or at least one person from the group, must look in the program or product $\mathcal{BP}$ and evaluate the quality characteristics we defined in our quality model, the evaluation could be in form of numerical value or different levels on aLickert scale.

7.2.8 Step 5: Computing Software Metrics over $\mathcal{BP}$

Using software metrics, we evaluate $\mathcal{BP}$ numerical values related to software internal attributes. There exist several software (commercial and open source) to compute software metrics on programs. For example, Ptidej includes an extension to compute several object-oriented software metrics on program models.

7.2.9 Step 6: Machine Learning Tools

JRip [91] as machine learning algorithm generate the relation between human evaluation of software quality (result from Step4) and value of software metrics (result from Step5). Weka's output is considered as set of RULE (an example is shown in Table 7.1) to be used for the evaluation of other software.

$$
\begin{aligned}
if \quad & (Metric_a \leq Value_a) \wedge (Metric_b \leq Value_b) && (7.1)\\
then \quad & (Q - Characteristic = Evaluation - Resualt_a) && (7.2)\\
else \quad & (Q - Characteristic = Evaluation - Resualt_b) && (7.3)
\end{aligned}
$$

Table 7.1: RULE for Q-Characteristicby evaluation the quality value of $\mathcal{BP}$

$$
\begin{aligned}
if \quad & (Metric_a \leq \frac{(U^{\mathcal{EP}}_{Metric_a} - L^{\mathcal{EP}}_{Metric_a})(Value_a - L^{\mathcal{BP}}_{Metric_a})}{U^{\mathcal{BP}}_{Metric_a} - L^{\mathcal{BP}}_{Metric_a}}) && (7.4)\\
& \wedge && (7.5)\\
& (Metric_b \leq \frac{(U^{\mathcal{EP}}_{Metric_b} - L^{\mathcal{EP}}_{Metric_b})(Value_b - L^{\mathcal{BP}}_{Metric_b})}{U^{\mathcal{BP}}_{Metric_b} - L^{\mathcal{BP}}_{Metric_b}}) && (7.6)\\
then \quad & (Q - Characteristic = Evaluation - Resualt_a) && (7.7)\\
else \quad & (Q - Characteristic = Evaluation - Resualt_b) && (7.8)
\end{aligned}
$$

Table 7.2: Adjustable Q-CharacteristicRULE for $\mathcal{EP}$

7.2.10 Step 7: Computing Software Metrics over $\mathcal{EP}$

Software metrics are used to assess the values of internal attributes over the $\mathcal{EP}$ in the same way as they were for the evaluation of $\mathcal{BP}$

7.2.11 Step 8: Adapting Metric

By using ratio over the values from Step7 and Step5, we can related the numerical values of Step7 with those of Step5. The following method will be used for relation evaluation:

Phase1. Finding the Max and Min value of each metrics in $\mathcal{EP}$.

Phase2. Finding the Max and Min value of same metrics we were compute on *Phase1* over the $\mathcal{BP}$.

Phase3. Analyzing the ratio for the values from *Phase1* plus values we have in RULE, we build a new RULE compatible with $\mathcal{EP}$: considering that upper range and lower range of metrics $Metric_b$ in $\mathcal{BP}$ is $U^{\mathcal{BP}}_{Metric_b}$ and $L^{\mathcal{BP}}_{Metric_b}$, then the new RULE for Q-Characteristicis presented in Table 7.2.

7.2.12 Step 9: Software Evaluation

Now, we can evaluate other programs ($\mathcal{EP}$) by applying the of adjusted RULE (from Step8) and software metrics evaluation over the $\mathcal{EP}$.

7.3 Implementation

We perform the following tasks to build Etiquette, a program for measuring software quality, based on our methodology.

7.3.1 Choosing an Evaluation Group

Evaluation begins by choosing the group of people (we call them $\mathcal{EG}$); these people once evaluate the small programs ($\mathcal{BP}$) and by using this evaluation, measurement of other program's quality are more useful for them ($\mathcal{EG}$).

We consider for our experience, groups of university's students who know enough about programming and are familiar with the basic concepts of software engineering.

7.3.2 Choosing a Scale

The evaluation uses 5 different levels on a Lickert scale and *N/A*for software characteristics or sub-characteristics.

7.3.3 Building a Quality Model

In Chapter 4, we introduced our quality model by considering those software characteristics which are related to design patterns.

7.3.4 Identifying a set of Base Programs ($\mathcal{BP}$)

We use the set of programs implementing design patterns from Kuchana's book [57] as base programs ($\mathcal{BP}$). Each program of this set implements design patterns from Gamma *et al.*'s book [38]. The source code which we considering them as $\mathcal{BP}$ is found on the following Web site: `http://www. crcpress.com/e_products/downloads/download.asp`.

7.3.5 $\mathcal{BP}$ Evaluation by $\mathcal{EG}$

By considering our model and the source code of $\mathcal{BP}$ programs, the $\mathcal{EG}$ provided values for each characteristics or sub-characteristics in our model. Table 7.3 present an example of evaluation done by $\mathcal{EG}$.

The results of $\mathcal{EG}$'s evaluation are stored in an XML file. Figure 7.6 presents an example for reserving the value of software quality items based on our model definition for the Abstract Factory design pattern.

	Quality Sub-characteristics and Characteristics										
Programs Group	Expendability	Simplicity	Generality	Modularity	Software Independence	Hardware Independence	Learnability	Understandability	Operability	Scalability	Robustness
Abstract Factory	*Excellent*	*Excellent*	*Good*	*Good*	*Fair*	*Fair*	*Good*	*Good*	*Good*	*Good*	*Good*

Table 7.3: Example from the result of $\mathcal{EG}$ evaluation

7.3.6 $\mathcal{BP}$ Design Pattern Identification

We need to identify classes playing roles in the design patterns in our programs. We analyse the programs and their micro-architectures using PADL, a meta-model to represent programs. Figure 7.7 presents an example from the format of presenting the classes, dividing by their roles in Abstract Factory design pattern.

7.3.7 Software Metrics

We choose size, filiation, coupling, cohesion, and complexity as internal attributes to measure quality. We use the metrics from Chidamber and Kemerer's study [19] mainly to measure these internal attributes, with additions from other metrics by Briand *et al.* [14], by Hitz and Montazeri [43], by Lorenz and Kidd [60], by Rosenberg and Hyatt [74] and by Tegarden *et al.* [87]. Table 7.4 presents the complete list of the metrics used for our evaluation, these metrics are implemented in POM and are used to evaluate $\mathcal{BP}$ as well as $\mathcal{EP}$.

7.3.8 $\mathcal{BP}$ Evaluation by Metrics

We apply POM, a framework for metrics definition and computation based on PADL [41], on the program models to compute the metric values. Figure 7.8 are presenting an example of metrics values which are calculated by POM.

Metrics Name	Metrics Description
ACAIC	Ancestor Class-Attribute Import Coupling
ACMIC	Ancestor Class-Method Import Coupling
AID	Average Inheritance Depth
CBO	Coupling Between Object Classes
CLD	Class-to-Leaf Depth
Cohesion Attributes	
Connectivity	
DCAEC	Descendant Class-Attribute Export Coupling
DCMEC	Descendant Class-Method Export Coupling
DIT	Depth of Inheritance Tree
ICHClass	Inheritance Complexity
LCOM1	Lack of Cohesion in Methods
LCOM2	Lack of Cohesion in Methods
LCOM5	Lack of Cohesion in Methods
NCM	Number of Class Methods in a class
NMA	Number of Methods Added
NMI	Number of Methods Inherited
NMO	Number of Methods Overridden
NOA	Number Of Ancestors
NOC	Number Of Children
NOD	Number Of Descendents
NOP	Number Of Public Attributes
SIX	Specialisation Index
WMC	Weighted Methods per Class

Table 7.4: POM Metrics List

7.3.9 Weka Appetizer

We create the data required by Weka, a machine-learning environment, separately for each quality characteristic and sub-characteristic of the quality model, using value attributed by $\mathcal{EG}$ and the values of the computed metrics. Each row presents the value of the evaluation of the quality characteristics and sub-Characteristics of our evaluation from $\mathcal{BP}$; and each column presents the values of the metrics. Figure 7.9 shows an example of the data for the Factory Method design pattern.

7.3.10 Linking Internal Attributes and Quality Characteristics

Using the JRip algorithm implemented in Weka, it is possible to find the relation between quality characteristics and values of the metrics. Figure 7.10 presents the rule associated with the Expandability quality characteristics.

7.3.11 Integrating our RULE

From JRip, we obtained the rules associated with a quality characteristics. We translate the rules to the format which we can use in Etiquette. Figure 7.11 presents an example of the integration of the RULE of Expandability in Etiquette.

7.3.12 $\mathcal{EP}$ Evaluation

We assess the quality of a program $\mathcal{EP}$ by applying POM to compute the same metric values we calculated for $\mathcal{BP}$. Figure 7.12 presents an example of the metric values, which are calculated by POM over a given $\mathcal{EP}$.

7.3.13 Applying the Rules

We adapt the metric values of the rules by computing the ratio between the minimum and maximin values of the metrics for the base programs, on the one hand, and each micro-architecture, on the other hand. Then, we compare the expected metric values in the adapted rules with the metric values computed for each micro-architecture and we update the RULE related to our software evaluation. We found several rules to assess the quality of software while considering the patterns used to design the architecture. Some results are shown in Table 7.5.

7.4 Conclusion

In this chapter, using the quality model, metrics, and study of the quality of previous sections, we build rules to assess the quality of programs while considering their architectures. We use a learning algorithm to infer rules from data collected on several program. We conclude that the rules highlight important quality characteristics of software. However, they are but a first step towards a complete and comprehensive assessment of software quality. In the next chapter, we detail our approach to assess the quality of software.

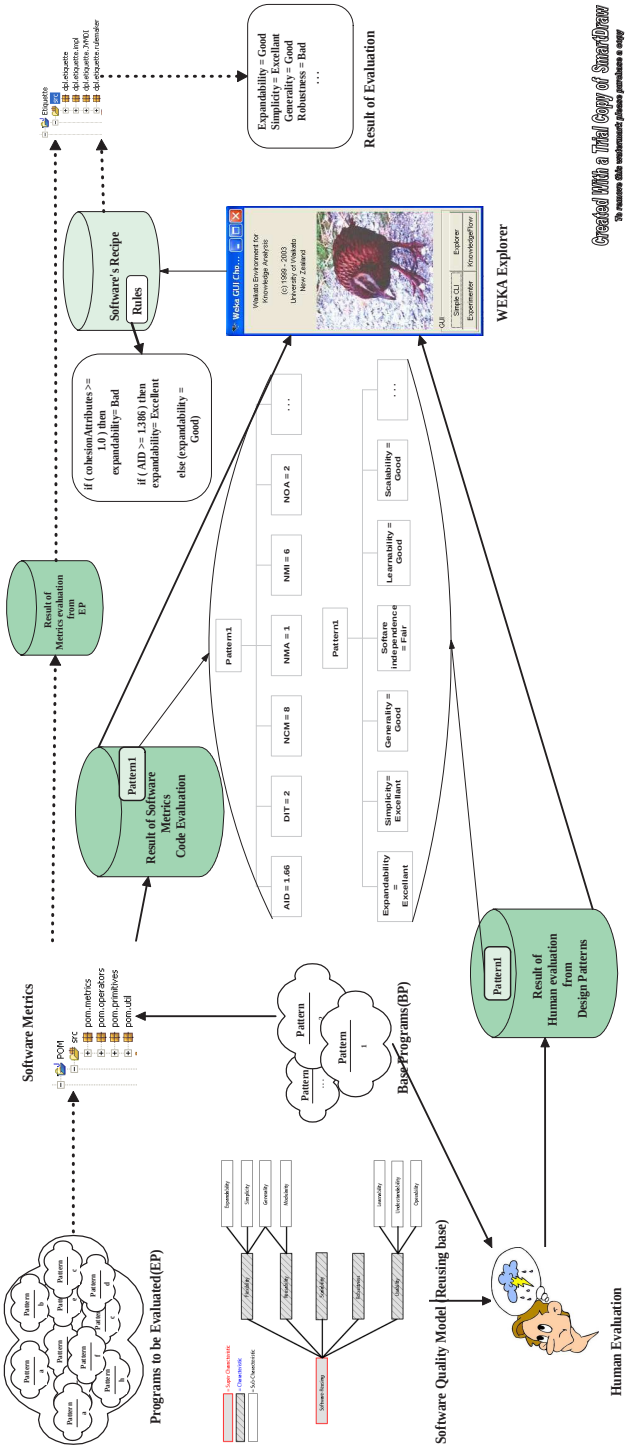

Figure 7.5: Etiquette General View

```
<designPattern name="Abstract Factory">
    <quality>
        <expandability          value="Excellent">  </expandability>
        <simplicity             value="Excellent">  </simplicity>
        <generality             value="Good">       </generality>
        <modularity             value="Good">       </modularity>
        <softwareIndependence   value="Fair">       </softwareIndependence>
        <hardwareIndependence   value="Fair">       </hardwareIndependence>
        <learnability           value="Good">       </learnability>
        <understandability      value="Good">       </understandability>
        <operability            value="Good">       </operability>
        <scalability            value="Good">       </scalability>
        <robustness             value="Good">       </robustness>
    </quality>
</designPattern>
```

Figure 7.6: Example of reserving the value of quality items in "Quality Pattern List.XML"

```
<designPattern name="Factory Method">
    <microArchitectures>
        <microArchitecture number="139">
            <roles>
                <products>
                    <product roleKind="AbstractClass">
                        <entity>src.FactoryMethod.After.ConsoleLogger</entity>
                    </product>
                </products>
                <concreteProducts>
                    <concreteProduct roleKind="Class">
                        <entity>src.FactoryMethod.After.FileLogger</entity>
                    </concreteProduct>
                </concreteProducts>
                <creators>
                    <creator roleKind="AbstractClass">
                        <entity>src.FactoryMethod.After.LoggerFactory</entity>
                    </creator>
                </creators>
                <concreteCreators>
                    <concreteCreator roleKind="Class">
                        <entity>src.FactoryMethod.After.LoggerTest</entity>
                    </concreteCreator>
                </concreteCreators>
            </roles>
        </microArchitecture>
    </microArchitectures>
</designPattern>
```

Figure 7.7: Example of presenting the different roles of Design Patterns in "Design Pattern List.XML"

```
<designPattern name="Factory Method">
    <microArchitectures>
        <microArchitecture number="139">
            <roles>
                <products>
                    <product roleKind="AbstractClass">
                        <entity kind="Class">src.FactoryMethod.After.ConsoleLogger</entity>
                        <properties>
                            <AID value="1.6666666666666667"/>
                            <CBO value="3.0"/>
                            <DIT value="2.0"/>
                            <NCM value="8.0"/>
                            <NMA value="1.0"/>
                            <NMI value="6.0"/>
                            <NMO value="1.0"/>
                            <NOA value="4.0"/>
                            <NOP value="4.0"/>
                            <SIX value="0.2857142857142857"/>
                            <WMC value="5.0"/>
                        </properties>
                        <relations>
                            <relation>
                                <target>java.io.PrintStream</target>
                                <properties>
                                    <CBO value="1.0"/>
                                    <oneWayCoupling value="1.0"/>
                                </properties>
                            </relation>
                        </relations>
                    </product>
                </products>
                <concreteProducts>
                    <concreteProduct roleKind="Class">
                        <entity kind="Class">src.FactoryMethod.After.FileLogger</entity>
                        <properties>
                            <AID value="1.6666666666666667"/>
                            <DIT value="2.0"/>
                            <cohesionAttributes value="1.0"/>
                            <ICHClass value="1.0"/>
                            <LCOM1 value="2.0"/>
                            <LCOM5 value="1.0"/>
                            <NCM value="9.0"/>
                            <NMA value="2.0"/>
                            <NMI value="6.0"/>
                            <NMO value="1.0"/>
                            <NOA value="7.0"/>
                            <NOP value="4.0"/>
                            <SIX value="0.25"/>
                            <WMC value="10.0"/>
                        </properties>
                        <relations>
                            <relation>
                                <target>src.BRIDGE.EncryptedMessage</target>
                                <properties>
                                    <CBO value="4.0"/>
                                    <oneWayCoupling value="3.0"/>
                                </properties>
                            </relation>
                        </properties>
                        ...
                    </concreteCreator>
                </concreteCreators>
            </roles>
        </microArchitecture>
    </microArchitectures>
</designPattern>
```

Figure 7.8: Example of metric values computed by POM over $\mathcal{BP}$

```
@RELATION learnability-Quality @ATTRIBUTE ACAIC REAL @ATTRIBUTE
ACMIC REAL @ATTRIBUTE AID REAL @ATTRIBUTE CBO REAL @ATTRIBUTE CLD
REAL @ATTRIBUTE DIT REAL @ATTRIBUTE connectivity REAL @ATTRIBUTE
DCAEC REAL @ATTRIBUTE DCMEC REAL @ATTRIBUTE ICHClass REAL @ATTRIBUTE
LCOM1 REAL @ATTRIBUTE LCOM2 REAL @ATTRIBUTE LCOM5 REAL @ATTRIBUTE
NCM REAL @ATTRIBUTE NMA REAL @ATTRIBUTE NMI REAL @ATTRIBUTE NMO REAL
@ATTRIBUTE NOA REAL @ATTRIBUTE NOC REAL @ATTRIBUTE NOD REAL
@ATTRIBUTE NOP REAL @ATTRIBUTE SIX REAL @ATTRIBUTE WMC REAL
@ATTRIBUTE cohesionAttributes REAL @ATTRIBUTE learnability
{'Excellent','Good','Fair','Bad','VeryBad'}

@DATA 0.0,0.0,1.3857142857142857,0.0,1.0,1.4,2.583333333...,Good
0.0,0.0,1.3333333333333333,0.0,0.0,1.3333333333333333,0....,Fair
0.0,0.0,1.3541666666666667,0.0,0.0,1.5,0.0,0.0,0.0,1.0,2...,Good
0.0,0.0,1.0,0.0,0.0,1.0,1.075,0.0,0.0,3.0,56.66666666666...,Fair
0.0,0.0,1.9885057471264367,0.0,0.0,2.0,0.0,0.0,0.0,1.0,2...,Fair
0.0,0.0,1.25,0.0,1.0,1.25,1.2333333333333334,0.0,0.0,0.0...,Good
0.0,0.0,1.9838709677419355,0.0,0.0,2.0,0.0,0.0,0.0,2.0,2...,Fair
0.0,0.0,1.25,0.0,1.0,1.25,1.25,0.0,0.0,0.0,6.0,0.0,1.5,7...,Fair
1.0,0.0,1.7863488372093024,0.0,1.0,1.8,0.0,0.0,0.0,1.333...,Good
0.0,0.0,1.4910714285714286,0.0,0.0,1.5,1.375,0.0,0.0,3.0...,Fair
0.0,0.0,1.0,0.0,0.0,1.0,1.0,0.0,0.0,0.0,4.0,0.0,0.75,9.0...,Good
0.0,0.0,1.3294573643410852,0.0,1.0,1.3333333333333333,1....,Fair
0.0,0.0,1.3333333333333333,0.0,1.0,1.3333333333333333,1....,Fair
0.0,0.0,1.0,0.0,0.0,1.0,1.6666666666666667,0.0,0.0,0.0,1...,Bad
0.0,0.0,1.246875,0.0,2.0,1.25,1.3333333333333335,2.0,4.0...,Fair
0.0,0.0,1.0,0.0,0.0,1.0,1.4444444444444444,0.0,0.0,0.0,5...,Good
0.0,0.0,1.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,2.0,0.0,1.818181...,Fair
0.0,0.0,1.0,0.0,0.0,1.0,0.5833333333333334,0.0,0.0,1.0,5...,Bad
0.0,0.0,1.740919242768009,0.0,1.0,1.75,1.241666666666666...,Fair
0.0,0.0,1.3333333333333333,0.0,1.0,1.3333333333333333,1....,Fair
0.0,0.0,1.3286384976525822,0.0,1.0,1.3333333333333333,1....,Bad
0.0,0.0,1.5,0.0,1.0,1.5,2.4166666666666665,0.0,0.0,6.0,8...,Good
0.0,0.0,1.9654761904761904,0.0,0.0,2.0,1.0,0.0,0.0,0.0,6...,Good
```

Figure 7.9: Example of "Factory Method.arff"; each row for Quality Characteristics and each column for metrics values

```
Test mode:    10-fold cross-validation

=== Classifier model (full training set) ===

JRIP rules:
===========

(cohesionAttributes >= 1) => expandability=Bad (4.0/1.0) (AID >=
1.385714) => expandability=Excellent (7.0/2.0)
 => expandability=Good (12.0/4.0)

Number of Rules : 3


Time taken to build model: 0.03 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances           6               26.087  %
Incorrectly Classified Instances        17               73.913  %
Kappa statistic                         -0.2532 Mean absolute error
0.2819 Root mean squared error                  0.4429
Relative absolute error                101.6733 %
Root relative squared error            119.7989 %
Total Number of Instances               23

=== Detailed Accuracy By Class ===

TP Rate   FP Rate   Precision   Recall  F-Measure   Class
  0         0.313      0         0         0        Excellent
  0.455     0.917      0.313     0.455     0.37     Good
  0         0          0         0         0        Fair
  0.333     0.05       0.5       0.333     0.4      Bad
  0         0          0         0         0        VeryBad

=== Confusion Matrix ===

 a b c d e   <-- classified as
 0 7 0 0 0 | a = Excellent
 5 5 0 1 0 | b = Good
 0 2 0 0 0 | c = Fair
 0 2 0 1 0 | d = Bad
 0 0 0 0 0 | e = VeryBad
```

Figure 7.10: Example of relation between characteristics and metric values

```
public final static Rule expandability_Bad =
    new Rule(
        "expandability_Bad",
            new FingerprintSet[] {
                new FingerprintSet(
                    new Fingerprint[] {
                        new Fingerprint("cohesionAttributes", Fingerprint.GEQ, 1.0),
}); public final static Rule expandability_Excellent =
    new Rule(
        "expandability_Excellent",
            new FingerprintSet[] {
                new FingerprintSet(
                    new Fingerprint[] {
                        new Fingerprint("AID", Fingerprint.GEQ, 1.3857142857142857),
}); public final static Rule expandability_Good =
    new Rule(
        "expandability_Good",
            new FingerprintSet[] {
                new FingerprintSet(
                    new Fingerprint[] {
                        new Fingerprint("cohesionAttributes", Fingerprint.LE, 1.0),
                new FingerprintSet(
                    new Fingerprint[] {
                        new Fingerprint("AID", Fingerprint.LE, 1.3857142857142857),
});
```

Figure 7.11: Example of RULE integration in Etiquette

```
<program type="Java">
    <name>6 - JHotDraw v5.1</name>
    <designPattern name="Adapter">
        <microArchitectures>
            <microArchitecture number="73">
                <roles>
                    <clients>
                        <client roleKind="AbstractClass">
                            <entity kind="Class">
                                CH.ifa.draw.standard.StandardDrawingView
                            </entity>
                            <properties>
                                <AID value="1.3333333333333333"/>
                                <CBO value="14.0"/>
                                <DIT value="3.0"/>
                                <cohesionAttributes value="0.08324552160168598"/>
                                <connectivity value="0.9961770623742455"/>
                                <ICHClass value="76.0"/>
                                <LCOM5 value="0.9718309843414267"/>
                                <NCM value="78.0"/>
                                <NMA value="34.0"/>
                                <NOP value="13.0"/>
                                <SIX value="1.5194805194805194"/>
                                <WMC value="312.0"/>
                            </properties>
                            <relations>
                                <relation>
                                    <target>CH.ifa.draw.applet.DrawApplet</target>
                                    <properties>
                                        <CBO value="26.0"/>
                                        <oneWayCoupling value="9.0"/>
                                    </properties>
                                </relation>
                                <relation>
                                    <target>
                                        CH.ifa.draw.application.DrawApplication
                                    </target>
                                    <properties>
                                        <CBO value="34.0"/>
                                        <oneWayCoupling value="9.0"/>
                                    </properties>
                                </relation>
                                ...
                                </relation>
                            </relations>
                        </concreteClass>
                    </concreteClasses>
                </roles>
            </microArchitecture>
        </microArchitectures>
    </designPattern>
</program>
```

Figure 7.12: Example of metrics value of metrics which calculated by POM over the $\mathcal{EP}$ (JHotDraw)

Micro-Architectures	Design Patterns	$LCOM5$	min_{LCOM5}	max_{LCOM5}	NOA	min_{NOA}	max_{NOA}	Applying RULE for learnability				
Subset of the micro-architectures in JHotDraw												
MA74	Command	1.07	0.50	1.63	29.35	1.00	164.00	$(LCOM5 \leq 1.16)$	$\wedge$	$(NOA \leq 62.30)$	$\Rightarrow$	*Good*
MA85	Singleton	0.67	0.67	0.67	1.00	1.00	1.00	$(LCOM5 \leq 0.00)$	$\wedge$	$(NOA \leq 0.00)$	$\Rightarrow$	*Fair*
MA91	Strategy	0.95	0.80	1.0	553.88	221.00	792.00	$(LCOM5 \leq 0.21)$	$\wedge$	$(NOA \leq 218.23)$	$\Rightarrow$	*Fair*
Subset of the micro-architectures in JUnit												
MA65	Composite	0.65	0.25	0.95	70.10	4.00	148.00	$(LCOM5 \leq 0.72)$	$\wedge$	$(NOA \leq 56.33)$	$\Rightarrow$	*Fair*
MA67	Iterator	0.92	0.83	0.99	30.67	1.00	48.00	$(LCOM5 \leq 0.17)$	$\wedge$	$(NOA \leq 18.38)$	$\Rightarrow$	*Fair*
MA68	Observer	0.90	0.66	1.03	112.43	1.00	191.00	$(LCOM5 \leq 0.38)$	$\wedge$	$(NOA \leq 74.32)$	$\Rightarrow$	*Fair*
MA70	Observer	0.83	0.83	0.83	11.00	11.00	11.00	$(LCOM5 \leq 0.00)$	$\wedge$	$(NOA \leq 0.00)$	$\Rightarrow$	*Fair*
Subset of the micro-architectures in Lexi												
MA8	Builder	0.95	0.93	0.97	7.75	1.00	12.00	$(LCOM5 \leq 0.03)$	$\wedge$	$(NOA \leq 4.30$	$\Rightarrow$	*Fair*
MA10	Observer	0.95	0.94	0.97	61.67	35.00	94.00	$(LCOM5 \leq 0.02)$	$\wedge$	$(NOA \leq 23.08)$	$\Rightarrow$	*Fair*
MA12	Singleton	0.99	0.99	0.99	2.00	2.00	2.00	$(LCOM5 \leq 0.00)$	$\wedge$	$(NOA \leq 0.00)$	$\Rightarrow$	*Fair*

Table 7.5: Example of applying the RULE quality model to subsets of JHotDraw, JUnit, and Lexi.

CHAPTER 8

CASE STUDY

We perform a case study to apply the previous approach to building and to applying a quality model considering program architectures. We use design patterns as a basis to build a quality model. We choose design patterns because they are now well-known constructs and have been studied extensively.

8.1 General Information

The following general information offer a synthetic view on our quality model.

8.1.1 Dependent Variables

The dependent variables in our quality model are quality characteristics. We choose these quality characteristics by studying the quality characteristics of the 23 design patterns in Gamma *et al.*'s book [38]. We study the literature on design patterns and identify 5 quality characteristics which decompose in 7 quality sub-characteristics which we consider as external attributes.

8.1.2 Independent Variables

The independent variables in our quality model are the internal attributes which we measure on programs. These internal attributes are similar to those in other quality models from the literature: Size, filiation, cohesion, coupling, and complexity.

8.1.3 Analysis Technique

We use a propositional rule learner algorithm, JRip. JRip is Weka—an open-source program collecting machine learning algorithms for data mining tasks [91]—implementation of the Ripper rule learner. It is a fast algorithm for learning "If–Then" rules. Like decision trees, rule learning algorithms are popular because the knowledge representation is easy to interpret.

8.2 Building a Quality Model

We perform the eight tasks identified in Subsection 7.1.2 to build a quality model considering program architectures based on design patterns.

8.2.1 Identifying Quality Characteristics

We consider a hierarchical model, because such model is more understandable [30] and because most of standard models are hierarchical, for examples [46] and [62].

Design patterns claim to bring reusability, understandability, flexibility, and modularity [38]. So, we add these quality characteristics to our quality model. Also, through our past experience, we add robustness and scalability (which define together software elegancy [35]) to our quality model.

8.2.2 Organising Quality Characteristics

We organise the quality characteristics and decompose these in sub-characteristics using definitions from *IEEE*, *ISO/IEC*, and several other models, such as McCall's, Boehm's, Firesmith's [32, 55, 66, 82].

Figure 4.2 presents our quality model to evaluate software quality related to software maintenance based on design patterns.

8.2.3 Choosing Internal Attributes

We choose size, filiation, coupling, cohesion, and complexity as internal attributes. We use the metrics from Chidamber and Kemerer's study [19] mainly to measure these internal attributes, with additions from other metrics by Briand *et al.* [14], by Hitz and Montazeri [43], by Lorenz and Kidd [60], and by Tegarden *et al.* [87].

The complete list of metrics used to measure internal attributes is: ACAIC, ACMIC, AID, CBO, CLD, cohesionAttributes, connectivity, DCAEC, DCMEC, DIT, ICHClass, LCOM1, LCOM2, LCOM5, NCM, NMA, NMI, NMO, NOA, NOC, NOD, NOP, SIX, and WMC.

8.2.4 Identifying Programs with Design Patterns

We use the set of programs implementing design patterns from Kuchana's book [57]. Each program of this set implements design patterns from Gamma *et al.*'s book [38]. This set of programs forms our base programs $\mathcal{BP}$.

8.2.5 Assessing the Quality of Design Patterns

We assess the quality characteristics of design patterns manually, using our quality model and the set $\mathcal{BP}$. Table 8.1 summaries our evaluation of the quality characteristics of the twenty-three design patterns.

	Quality Sub-characteristics and Characteristics								
Design Patterns	Expendability	Simplicity	Generality	Modularity	Learnability	Understandability	Operability	Scalability	Robustness
Abs. Fact.	E	E	G	G	G	G	G	G	G
Builder	G	G	F	F	F	G	F	G	G
Fact. Met.	P	P	F	G	G	G	G	G	G
Prototype	E	G	F	G	F	G	F	E	G
Singleton	P	B	F	E	F	F	F	G	G
Adapter	F	F	P	G	G	F	F	G	F
Bridge	G	F	G	G	F	F	G	G	G
Composite	F	F	F	F	F	G	F	F	G
Decotator	E	E	G	F	G	G	G	G	F
Façade	G	G	G	G	F	G	F	F	F
Flyweight	P	P	F	G	G	P	F	G	G
Proxy	G	P	F	G	F	P	G	G	F
Chain of Res.	G	G	G	P	F	F	G	P	F
Command	G	P	F	F	P	B	G	G	G
Interpreter	G	F	G	F	F	F	G	G	F
Iterator	E	E	G	F	G	F	F	G	G
Mediator	G	F	G	G	F	F	G	G	F
Memento	G	F	F	B	P	F	G	F	P
Observer	E	G	E	F	F	G	G	G	G
State	G	G	F	P	F	B	G	G	F
Strategy	G	F	P	F	P	P	F	P	F
Tem. Met.	E	G	F	F	G	G	G	G	G
Visitor	E	G	G	F	G	P	F	G	F

Table 8.1: Design patterns quality characteristics in $\mathcal{BP}$ (E = Excellent, G = Good, F = Fair, P = Poor, and B = Bad)

8.2.6 Computing Metrics

The metrics we chose in task 3 to measure the internal attributes of programs are all class-based metrics. Thus, we need first to compute the metric values and second to adapt the metric values to the micro-architectures.

We analyse the programs and their micro-architectures using PADL, a meta-model to represent programs. Then, we apply POM, a framework for metrics definition and computation based on PADL [41], on the program models to compute the metric values.

Then, we adapt the class-based metric values to the micro-architectures. For a given metric, we use the average of its values on all the classes forming a micro-architecture. However, average is not a good representative of the metric values for the micro-architecture. Indeed, we should compute and study the variance of the metric values to get a better

$$
\begin{array}{rl}
if & (LCOM5 \leq 1.1) \wedge (NOA \leq 33.25) \\
then & (Learnability = Good) \\
else & (Learnability = Fair)
\end{array}
$$

Table 8.2: Rule for learnability

understanding of the distribution of the metric values. Variance indicates how much each of the metric values of the classes in the micro-architecture deviates from the mean. However, for the current exploratory study, we keep the average to allow a better analysis of the resulting rules.

8.2.7 Linking Internal Attributes and Quality Characteristics

We use a machine learning technique to infer rules linking the quality characteristics of the quality model and the metric values.

We use the JRip algorithm to find the rules between quality characteristics and values of the metrics. The rule in Table 8.2 is the rule associated with the learnability quality characteristics, when applying JRip in the metric values and the base programs from tasks 3, 4 and 5. It shows that the learnability quality characteristics is related to the NMI and NOP metrics more than to any other metric.

We do not introduce here all the rules found for the different quality sub-characteristics and characteristics in our model for lack of space. The rules are specific to the current case study but help in illustrating the advantages and limitations of our approach.

8.2.8 Validating the Quality Model

We use the leave-one-out method [83] for cross-validating the rules built for our quality model by JRip.

8.3 Applying the Quality Model

We apply the quality model built in the previous Subsection 8.2 to JHotDraw (we only apply our model on a subset of the micro-architectures for lack of space), JUnit, and Lexi programs. For lack of space, we only apply the learnability Rule 1 of the quality model. Rule 1 has been built in task 7 in Subsection 8.2 with $min_{LCOM5} = 0.75$, $max_{LCOM5} = 1.82$, $min_{NOA} = 1.00$, and $max_{NOA} = 86.00$.

JHotDraw is a Java GUI framework for technical and structured graphics. It has been developed originally by Erich Gamma and Thomas Eggenschwiler as a "design exercise" but is now a full-fledge framework. Its design relies heavily on some well-known design patterns. JUnit is a regression testing framework written by Erich Gamma and Kent Beck. It is used to implements unit tests in Java. Both Lexi is a Java-based Word Processor. It has been developed by Matthew Schmidt and Brill Pappin originally. programs are open-source and hosted on SourceForge.

Applying our quality model requires to identify in the two programs the micro-architectures similar to some design patterns and the related classes. We use PMARt, an XML database which contains micro-architectures similar to design patterns in several programs [41], to extract the micro-architectures in JHotDraw, JUnit, and in Lexi. We also use PMARt, along with PADL and POM, to compute all the metrics defined in our quality model over each class of the known micro-architectures. We develop Etiquette, a tool to compute the average and the variance of the class-based metric values for each micro-architecture.

Now, we follow the four tasks from Subsection 7.1.3.

8.3.1 Identifying Micro-Architectures

JHotDraw uses 11 different design patterns in 21 micro-architectures: Adapter, Command, Composite, Decorator, Factory Method, Observer, Prototype, Singleton, State, Strategy, and Template Method. JUnit contains 8 micro-architectures similar to 5 different design patterns: Composite, Decorator, Iterator, Observers, and Singletons. Lexi contains 5 micro-architectures similar to the Builder, Observer, and Singleton design patterns. Table 8.3 summarises the micro-architectures.

8.3.2 Measuring Internal Attributes

For each micro-architecture identified in JHotDraw or in JUnit or in Lexi, we use PADL and POM to compute the class-based metric values and Etiquette to compute the micro-architecture-based metric values (using average). Table 8.3 presents the data for each micro-architecture for the LCOM5 and NOA metrics.

8.3.3 Applying the Rules

We adapt the metric values in the rule in Table 8.2 by computing the ratio between the minimum and maximin values of the LCOM5 and NOA metrics for the base programs on the one hand, and each micro-architecture on the other hand. Table 8.3 also displays

Micro-Architectures	Design Patterns	$LCOM5$	$minLCOM5$	$maxLCOM5$	NOA	$minNOA$	$maxNOA$	Rule for learnability		
Subset of the micro-architectures in JHotDraw										
MA74	Command	1.07	0.50	1.63	29.35	1.00	164.00	$(LCOM5 \leq 1.16) \wedge (NOA \leq 62.30)$	$\Rightarrow$	$Good$
MA85	Singleton	0.67	0.67	0.67	1.00	1.00	1.00	$(LCOM5 \leq 0.00) \wedge (NOA \leq 0.00)$	$\Rightarrow$	$Fair$
MA91	Strategy	0.95	0.80	1.0	553.88	221.00	792.00	$(LCOM5 \leq 0.21) \wedge (NOA \leq 218.23)$	$\Rightarrow$	$Fair$
Subset of the micro-architectures in JUnit										
MA65	Composite	0.65	0.25	0.95	70.10	4.00	148.00	$(LCOM5 \leq 0.72) \wedge (NOA \leq 56.33)$	$\Rightarrow$	$Fair$
MA66	Decorator	0.65	0.25	0.90	135.41	49.00	176.00	$(LCOM5 \leq 0.67) \wedge (NOA \leq 49.68)$	$\Rightarrow$	$Fair$
MA67	Iterator	0.92	0.83	0.99	30.67	1.00	48.00	$(LCOM5 \leq 0.17) \wedge (NOA \leq 18.38)$	$\Rightarrow$	$Fair$
MA68	Observer	0.90	0.66	1.03	112.43	1.00	191.00	$(LCOM5 \leq 0.38) \wedge (NOA \leq 74.32)$	$\Rightarrow$	$Fair$
MA69	Observer	0.83	0.83	0.83	1.00	1.00	1.00	$(LCOM5 \leq 0.00) \wedge (NOA \leq 0.00)$	$\Rightarrow$	$Fair$
MA70	Observer	0.83	0.83	0.83	11.00	11.00	11.00	$(LCOM5 \leq 0.00) \wedge (NOA \leq 0.00)$	$\Rightarrow$	$Fair$
MA71	Singleton	0.00	0.00	0.00	1.00	1.00	1.00	$(LCOM5 \leq 0.00) \wedge (NOA \leq 0.00)$	$\Rightarrow$	$Fair$
MA72	Singleton	0.00	0.00	0.00	1.00	1.00	1.00	$(LCOM5 \leq 0.00) \wedge (NOA \leq 0.00)$	$\Rightarrow$	$Fair$
Subset of the micro-architectures in Lexi										
MA8	Builder	0.95	0.93	0.97	7.75	1.00	12.00	$(LCOM5 \leq 0.03) \wedge (NOA \leq 4.30)$	$\Rightarrow$	$Fair$
MA9	Observer	0.95	0.94	0.97	9.50	1.00	18.00	$(LCOM5 \leq 0.02) \wedge (NOA \leq 6.65)$	$\Rightarrow$	$Fair$
MA10	Observer	0.95	0.94	0.97	61.67	35.00	94.00	$(LCOM5 \leq 0.02) \wedge (NOA \leq 23.08)$	$\Rightarrow$	$Fair$
MA11	Singleton	1.01	1.01	1.01	1.00	1.00	1.00	$(LCOM5 \leq 0.00) \wedge (NOA \leq 0.00)$	$\Rightarrow$	$Fair$
MA12	Singleton	0.99	0.99	0.99	2.00	2.00	2.00	$(LCOM5 \leq 0.00) \wedge (NOA \leq 0.00)$	$\Rightarrow$	$Fair$

Table 8.3: Data and rules when applying the quality model to a subset of JHotDraw, JUnit, and Lexi

the adapted rules for all the micro-architectures. We compare the expected metric values in the adapted rules with the metric values computed for each micro-architecture.

Table 8.3 presents the results for adapting the learnability rule in Table 8.2. We computed the average, the minimum, and the maximum values of the LCOM5 and NOA metrics for each program, JUnit and Lexi. We adapted the rule from the minimum and maximum values of the base programs and of JHotDraw, JUnit, and Lexi. The last column shows the adapted rules and the results of applying the rules.

The first line of the table shows an example of applying the learnability rule to a micro-architecture similar to the Command design pattern. The outcome of the rule states that this particular implementation of the Command design pattern has a Good learnability.

However, the quality model obtained is unsatisfactory for many reasons. First, the size of the base programs used to create the quality model renders the rule uninteresting in many cases. In particular, we do not have sufficient data yet but the assess the learnability of JUnit and of Lexi as Fair.

Second, adapting the rule when there is one metric value only, see for example the micro-architecture MA5 in JUnit, does not provide interesting information because the adapted threshold of the learnability rule is always inferior to the maximum (and unique) value. Adaptation requires a range or more accurate rules (based on a minimum and a maximum thresholds) to be efficient.

Third, we do not distinguish in the micro-architecture between code which plays a role in the design pattern and code which does not. Considering all the metric values, potentially for "dead" code, has an impact on the results certainly.

Moreover, learnability is a human-related quality characteristic. Thus, it is difficult to assess intrinsically because it depends on the individuals performing the evaluation highly. Thus, we need to perform more evaluations to obtain an accurate rule.

8.4 Discussion

8.4.1 Human Factor

We attempt to place *people*, software developers and maintainers alike, in the center of our work because software building and maintenance remains human-intensive activities. Indeed, software are still products of creation and imagination—even of art [8] with no two programs alike—rather than manufactured products. Work on manufacturing programs, such as software product lines [13] and software factories [23], exist but are yet to be put in practice. Thus, in our quality model, we attempt to assess the quality of programs with subjective quality characteristics that relate to the work of maintainers, such as

expandability, simplicity, learnability, rather than with objective quality characteristics, such as fault proneness [16].

Also, the quality model, its rules, depends on the experience of the software engineers who built the model, because the quality model is built from their subjective evaluation of the quality of patterns. On the hand, the quality model is tailored for the group of software engineers who built it. On the other hand, the quality model must be applied with caution out of this group.

8.4.2 Patterns and Architectures

In our case study, we use design patterns because they are now well-known constructs. However, design patterns are but a step in considering program architectures as wholes rather than as aggregates of smaller pieces. With design patterns, we use bigger parts than classes to assess the quality characteristics of programs. It is similar to assessing the quality of a painting by looking at many small squares extracted from the painting, such as in Figure 7.3, on the right. We need to further develop our work to take in account the actual architectural *styles* [79] of programs as wholes.

8.4.3 Data

Our work suffers from a lack of data on the subjective, or *perceived*, quality characteristics of patterns and of programs to build our quality model. We build a first quality model using 23 programs implementing the 23 design patterns from Gamma *et al.* [38]. We need to pursue our study to enrich our data set and to build more accurate quality models. Indeed, we do not have sufficient data yet to produce but a case study of a quality model considering program architectures.

We need more data on the subjective evaluation of the quality characteristics of patterns to assign consensual values to their characteristics. We also need more data on the subjective evaluation of the quality characteristics of programs with patterns, $\mathcal{BP}$, to create a quality model generalisable to other programs. We plan to produce a survey on the subjective perception of the quality characteristics of design patterns and of programs and to post this survey on the Internet to get as many answers as possible.

8.4.4 Generalisation

We build our quality model based on patterns, thus we apply our quality model on micro-architectures rather than on programs. It is always possible to extract a micro-architecture from a program architecture, for example using design pattern identification techniques, feature identification techniques, or arbitrarily, and then to apply our quality

model on this micro-architecture. Also, the micro-architecture can be of arbitrary size. Thus, we believe that it is possible to apply our quality model on any programs, though the use average and of ratio.

However, we are yet unsure of the *legitimacy* of the generalisation of our quality model (or of any quality models for that matter) to other *similar* programs. Indeed, we believe that there are major differences between studies, for example, on people and on programs.

First, people share many common characteristics that are intrinsic to the human race and that make everybody similar yet not identical. Many theories exist to predict and to explain how a human being *is*, in general and in particular. These theories can also be used to distinguish between human being and other beings, comparing for examples, the number of members, the physiology of organs.

Unfortunately, similar theories predicting and explaining and comparing programs do not exist yet. To our best knowledge, there is no general consensus on what programs *are*, except at the very basic level, for example, a set of lines of code written in a programming language (which is similar to saying that a being is a human being because it is composed of cells).

Thus, each program is unique and there do not exist satisfying classifications of programs. This lack of classification impedes the successful development of quality models and their applications because we cannot ensure that a quality model built from a set of programs can be applied to another set of programs *legitimately*. We believe that a possible classification of programs could stem from the use or lack of use of patterns in the program architectures.

Second, programs are discrete by nature. Unlike *physical* artifacts, such as bridges, programs are made of discrete parts: Classes, fields, methods, functions, which compose programs. Bridges also are composed of parts: Bolts, rails, concrete, which can be modeled mathematically and which properties as a whole can be described, for example with finite-elements method. To our best knowledge, models do not yet exist to describe programs, which remain discrete by their very nature. Quality models are a step towards building theories on programs, because these models help in evaluating and in predicting certain characteristics of software artifacts. However, they require to consider programs as wholes rather than as aggregates of classes. Our approach is a step towards building a theory relating the architectures of programs with their quality characteristics.

CHAPTER 9

CONCLUSION

In this dissertation, we proposed a thorough study of quality characteristics in software products. We present different quality models, which decompose in hierarchical and non-hierarchical models, such as McCall, Boehm, FURPS, *ISO*, Dromey, Star model and Bayesian Belief Networks. We complete the models with definitions of quality characteristics and interrelationships among them.

We associate software metrics with attributes defined in the different models, like Adaptability, Completeness, Complexity, Conciseness, Correctness, Efficiency, Expendability, Generality, Hardware-independence, Indicesability, learnability, Modularity, Maturity index, Operability, Portability, Readability, Reliability, Robustness, Scalability, Simplicity, Software independence, Structuredness, Traceability, Understandability, and Usability.

Then, we introduce a model to assess the ability of design patterns claim to increase the Reusability, Flexibility, Modularity, Understandability and Software Elegancy. At last we have an evaluation of twenty-three design patterns based on Gamma *et al.* [38].

Software quality models must state clearly their target user's and define an supplementary layer of characteristics ("super"-characteristics) to be more useful and comparable.

Software quality models must take into account other aspects of software such as their performance, runtime adequacy, and architecture (for example, through the evaluation of design patterns).

We presented an approach to building and to applying quality models considering program architectures. This approach is based on patterns, for example design patterns, to consider program architectural quality characteristics rather than classes or couples thereof only. The use of patterns in building and in applying a quality model brings an extra level of abstraction to the quality model.

In particular, we circumvent the lack of architectural metrics by computing the average of class-based metric values from classes participating in a pattern.

Also, we adapt the rules of a quality model to different programs using a ratio-based technique. Computing the ratio allows applying the rules built from a set of programs to any other program.

We introduced a case study of our approach to build a quality model based on design patterns and to apply, with success, the resulting quality model on the JUnit and Lexi

programs.

Our first contribution is a complete study of the state-of-the-art on quality model and design patterns. Our second contribution is a quality model with associated metrics. Our third contribution is a thorough study of the quality characteristics of the 23 design patterns from Gamma *et al.* [38]. Our fourth and final contribution is a quality model built using learning algorithm to assess the quality of programs while considering the design patterns used in its architecture.

For future research we plan to continue our study of the JHotDraw, JUnit, and Lexi programs in details, considering "dead" code and assessing the validity of each rule. We also plan to realise a detailed survey of the quality characteristics of design patterns through the Internet to collect as much data as possible. Using the results of this survey, we shall then apply our approach to refine the rules built for the quality model. The rules built for the quality model are a first step towards a theory on quality and design patterns. We shall study the relation between the rules and well-known principles of software engineering to study the intrinsic advantages of design patterns.

BIBLIOGRAPHY

[1] Anthony Aaby. Software life cycle models, 2003. http://cs.wwc.edu/ãabyan/435/models.html.

[2] Anthony Aaby. Software maintenance: Activities required to provide cost-effective support to a software system, 2003. http://cs.wwc.edu/ãabyan/435/Maintenance.html.

[3] Anthony A. Aaby. *Software: A Fine Art.* Creative Commons Attribution-NonCommercial License, January 2004.

[4] Brad Appleton. An introduction to the history, origins, and essential concepts and terminology of software patterns, May 1997.

[5] Lowell Jay Arthur. *Software Evolution: The Software Mainenance Challenge.* John Wiley and Sons, 1^{st} edition, February 1988.

[6] Osman Balci. Credibility assessment of simulation results. In James R. Wilson, James O. Henriksen, and Stephen D. Roberts, editors, *proceedings of the* 18^{th} *Conference on Winter Simulation*, pages 38–44. ACM Press, December 1986.

[7] Victor R. Basili. The experimental paradigm in software engineering. In H. Dieter Rombach, Victor R. Basili, and Richard W. Selby, editors, *proceedings of the international workshop on Experimental Software Engineering Issues: Critical Assessment and Future Directions*, pages 3–12. Springer Verlag, September 1992.

[8] Victor R. Basili. The role of experimentation in software engineering: Past, current, and future. In H. Dieter Rombach, editor, *proceedings of the* 18^{th} *International Conference on Software Engineering*, pages 442–449. IEEE Computer Society Press, March 1996.

[9] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional, 1^{st} edition, October 1999.

[10] Nigel Bevan. Quality in use: Meeting user needs for quality. *Journal of Systems and Software*, 49(1):89–96, December 1999.

[11] James M. Bieman and Byung-Kyoo Kang. Measuring design-level cohesion. *Transactions on Software Engineering*, 24(2):111–124, February 1998.

[12] Barry W. Boehm, James R. Brown, and Myron Lipow. Quantitative evaluation of software quality. In Raymond T. Yeh and C. V. Ramamoorthy, editors, *proceedings of the* 2^{nd} *International Conference on Software Engineering*, pages 592–605. ACM Press, October 1976. http://portal.acm.org.

[13] Jan Bosch. Product-line architectures in industry: A case study. In *proceedings of the* 21^{st} *International Conference on Software Engineering.* IEEE Computer Society Press, May 1999.

[14] Lionel Briand, Prem Devanbu, and Walcelio Melo. An investigation into coupling measures for C++. In W. Richards Adrion, editor, *proceedings of the* 19^{th} *International Conference on Software Engineering*, pages 412–421. ACM Press, May 1997.

[15] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A unified framework for coupling measurement in object-oriented systems. *Transactions on Software Engineering*, 25(1):91–121, January–February 1999.

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

[17] Luigi Buglione and Alain Abran. Geometrical and statistical foundations of a three-dimensional model of software performance. *Advances in Engineering Software*, 30(12):913–919, December 1999.

[18] David N. Card and Robert L. Glass. *Measuring software design quality.* Prentice-Hall, Inc., 1990.

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

[20] Space Applications Corporation. Software reuse metrics. Technical report, NIST, Jan 1993.

[21] Lisa Crispin. Is quality negotiable? *STARWest 2001 conference*, jul 2001.

[22] Philip B. Crosby. *Quality Is Free.* Signet, reissue edition, January 1980.

[23] Michael A. Cusumano. *Japan's Software Factories: A Challenge to U.S. Management.* Oxford University Press, 1^{st} edition, March 1991.

[24] Data and Object Factory. Software design patterns, June 2002.

[25] Dennis de Champeaux. *Object-Oriented Development Process and Metrics.* Prentice Hall, 1^{st} edition, September 1996.

[26] W. Edwards Deming. *Out of the Crisis.* The MIT Press, 1^{st} edition, August 2000.

[27] Fausto Distante, Mariagiovanna G. Sami, and Giancarlo Storti Gajani. A general configurable architecture for WSI implementation for neural nets. In *proceedings of the* 2^{nd} *International Conference on Wafer Scale Integration*, pages 116–123. IEEE Computer Society Press, January 1990.

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

[29] Norman E. Fenton. *Software Metrics: A Rigorous Approach.* International Thomson Computer Press, 4^{th} edition, November 1996.

[30] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics A Rigorous and Practical Approach.* PWS Publishing Company, 2^{nd} edition, October 1997.

[31] Donald Firesmith. A hard look at quality management software, April 2004.

[32] Donald G. Firesmith. Common concepts underlying safety, security, and survivability engineering. Technical Report CMU/SEI-2003-TN-033, Software Engineering Institute, December 2003.

[33] Ronan Fitzpatrick. Software quality definitions and strategic issues. Master's thesis, School of Computing, Staffordshire University, April 1996.

[34] Christiane Floyd. *Human Questions in Computer Science*, chapter 1, pages 15–27. Springer Verlag, March 1992.

[35] Center for Software Engineering. *OO Analysis and Design: Modeling, Integration, Abstraction.* University of Southern California, January 2002. http://sunset.usc.edu/classes/cs577b_2002/EC/03/EC-03.ppt.

[36] John E. Gaffney. Metrics in software quality assurance. In Beth Levy, editor, *proceedings of the ACM'81 Conference*, Association for Computing Machinery, pages 126–130. ACM press, March 1981.

[37] Peter Baer Galvin. Storage consolidation – part, August 2002.

[38] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1^{st} edition, 1994.

[39] Alan Gillies. *Software Quality: Theory and Management*. Chapman & Hall, 1^{st} edition, September 1992.

[40] Robert L. Glass. The software–research crisis. *Software*, 11(6):42–47, November 1994.

[41] Yann-Gaël Guéhéneuc, Houari Sahraoui, and Farouk Zaidi. Fingerprinting design patterns. In Eleni Stroulia and Andrea de Lucia, editors, *proceedings of the* 11^{th} *Working Conference on Reverse Engineering*, pages 172–181. IEEE Computer Society Press, November 2004.

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

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

[44] Larry Hyatt and Linda Rosenberg. A software quality model and metrics for risk assessment. In *proceedings of the* 8^{th} *Annual Software Technology Conference*. European Space Agency, April 1996.

[45] IEEE. Software engineering collection 982.1 standard dictionary of measures to produce reliable. *Institute of Electrical and Electronics Engineers*, 1996.

[46] ISO/IEC. *Information Technology – Software Product Evaluation – Quality Characteristics and Guidelines for their Use*, December 1991. ISO/IEC 9126:1991(E).

[47] ISO/IEC. Standard 14598-1, 1999.

[48] ISO/IEC. Standard 9126 – part 1, 2, 3: Quality model, 2001.

[49] Prasad Jogalekar and Murray Woodside. Evaluating the scalability of distributed systems. *Transactions on Parallel and Distributed Systems*, 11(6):589–603, June 2000.

[50] J.M. Juran and Frank M. Gryna. *Juran's Quality Control Handbook*. Mcgraw-Hill, 4^{th} edition, August 1988.

[51] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley, 2^{nd} edition, December 2002.

[52] Byung-Kyoo Kang and James M. Bieman. Design-level cohesion measures: Derivation comparison and applications. In Carl K. Chang and Sukho Lee, editors, *proceedings of the* 20^{th} *Computer Software and Applications Conference*, pages 92–97. IEEE Computer Society Press, August 1996.

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

[54] Souheil Khaddaj and Gerard Horgan. The evaluation of software quality factors in very large information systems. *Electronic Journal of Information Systems Evaluation*, 7(1):43–48, January 2004. Series: Academic Conferences Limited.

[55] Khashayar Khosravi and Yann-Gaël Guéhéneuc. A quality model for design patterns. Technical Report 1249, University of Montreal, September 2004.

[56] Barbara Kitchenham and Shari Lawrence Pfleeger. Software quality: The elusive target. *Software*, 13(1):12–21, January 1996.

[57] Partha Kuchana. *Software Architecture Design Patterns in Java.* Auerbach Publications, 1^{st} edition, April 2004.

[58] D. Richard Kuhn, William J. Majurski, Wayne McCoy, and Fritz Schulz. *Open Systems Software Standards in Concurrent Engineering.* Advances in Control and Dynamic Systems. Academic Press, June 1994.

[59] Gunther Lehmann and Bernhard Wunder andKlaus D. Muller-Glaser. Basic concepts for an HDL reverse engineering tool-set. In *proceedings of the* 9^{th} *International Conference on Computer-Aided Design*, IEEE-CS : Computer Society - IEEE-CAS : Circuits & Systems - SIGDA: ACM Special Interest Group on Design Automation, pages 134–141. IEEE Computer Society Press, November 1996.

[60] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics.* Prentice Hall, 1^{st} edition, July 1994.

[61] Salvatore Mamone. Standard for software maintenance. *Software Engineering Notes*, (19):75–76, January 1994.

[62] James A. McCall. Quality factors. *Encyclopedia of Software Engineering*, 1-2:958–ff, December 2001.

[63] Snoeck Monique, Wijsen Jozef, and Dedene Guido. Formal specifications in object oriented analysis: A comparative view. In *proceedings of the* 1^{st} *Workshop on Evaluation of Modeling Methods in Systems Analysis and Design.* Springer-Verlag, May 1996.

[64] Martin Neil and Norman Fenton. Predicting software quality using bayesian belief networks. In *proceedings of the* 12^{st} *Annual Software Engineering Workshop*, pages 217–230. NASA Goddard Space Flight Centre, December 1996.

[65] Martin Neil, Norman Fenton, and Lars Nielsen. Building large-scale bayesian networks. *The Knowledge Engineering Review*, 15(3):257–284, September 2000.

[66] CBR Online. Scalability from the edge. *Computer Business review Online, CBR Online*, Jun 2002.

[67] Gerard O'Regan. *A Practical Approach to Software Quality.* Springer, 1^{st} edition, June 2002.

[68] Maryoly Ortega, María A. Perez, and Teresita Rojas. A systemic quality model for evaluating software products. In *proceedings of the* 6^{th} *World Multiconference on Systemics, Cybernetics, and Informatics.* IIIS, June 2002.

[69] Sassan Pejhan, Alexandros Eleftheriadis, and Dimitris Anastassiou. Distributed multicast address management in the global internet. *Journal of Selected Areas in Communications*, 13(8):1445–1456, October 1995.

[70] Peter Petersen and Tom Schotland. *Win32 and Real Time*, April 1999. http://www.circuitcellar.com/library/print/0499/Schotland105/6.htm.

[71] Shari Lawrence Pfleeger. *Software Engineering Theory and Practice.* Prentice Hall, 2^{nd} edition, February 2001.

[72] Roger S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, 2^{nd} edition, June 1992.

[73] Roger S. Pressman and Roger Pressman. *Software Engineering: A Practitioner's Approach with Bonus Chapter on Agile Development.* McGraw-Hill Science, 5^{th} edition, December 2003.

[74] Linda H. Rosenberg and Lawrence E. Hyatt. Software quality metrics for object-oriented environments, April 1997. http://satc.gsfc.nasa.gov/support/CROSS_APR97/oocross.html.

[75] James Rumbaugh. Relations as semantic constructs in an object-oriented language. In Norman Meyrowitz, editor, *proceedings of the* 2^{nd} *conference on Object-Oriented Programming Systems, Languages and Applications*, pages 466–481. ACM Press, October 1987.

[76] Houari A. Sahraoui, Mounir Boukadoum, and Hakim Lounis. Building quality estimation models with fuzzy threshold values. *L'Objet*, 17(4):535–554, March 2001.

[77] Wayne J. Salamon and Dolores R. Wallace. Quality characteristics and metrics for reusable software. Technical Report 5459, National Institute of Standards and Technology, May 1994. (Preliminary Report).

[78] Joc Sanders and Eugene Curran. *Software Quality: A Framework for success in software Development and Support.* Addison-Wesley, 1^{st} edition, August 1995.

[79] Mary Shaw and Paul Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *proceedings of the* 21^{st} *international Computer Software and Applications Conference*, pages 6–13. IEEE Computer Society Press, August 1997.

[80] Walter Shewarth. *The Economic Control of Manufactured Products.* Amer Society for Quality, 50^{th} anniversary commemorative issue edition, December 1980.

[81] Abraham Silberschatz and Peter B. Galvin. *Operating system concepts.* Addison-Wesley, 4^{th} edition, June 1994.

[82] Connie U. Smith and Lloyd G. Williams. *Introduction to Software Performance Engineering*, chapter 1. Addison-Wesley, November 2001.

[83] M. Stone. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society*, 36:111–147, 1974. Series B: Statistical Methodology.

[84] Wolfgang B. Strigel, Geoff Flamank, and Gareth Jones. What are software metrics?, 1992.

[85] Ladan Tahvildari. Assessing the impact of using design pattern based systems. Master's thesis, University of Waterloo, 1999.

[86] Michiaki Tatsubori and Shigeru Chiba. Programming support of design patterns with compile-time reflection. In *proceedings of the* 1^{st} *OOPSLA Workshop on Reflective Programming in C++ and Java*, pages 56–60. ACM Press, October 1998.

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

[88] John B. Tran, Michael W. Godfrey, Eric H. S. Lee, and Richard C. Holt. Architectural repair of open source software. In *proceedings of the* 8^{th} *International Workshop on Program Comprehension*, pages 48–57. IEEE Computer Society Press, June 2000.

[89] Jan Tretmans and Peter Achten. Quality of information systems. *Kwaliteit van Informatiesystemen*, September 1999.

[90] Peter Wendorff. Assessment of design patterns during software reengineering: Lessons learned from a large commercial project. In Pedro Sousa and Jürgen Ebert, editors, *proceedings of* 5^{th} *Conference on Software Maintenance and Reengineering*, pages 77–84. IEEE Computer Society Press, March 2001.

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

[92] Murray Wood, John Daly, James Miller, and Marc Roper. Multi-method research: An empirical investigation of object-oriented technology. *Journal of Systems and Software*, 48(1):13–26, August 1999.

[93] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In Joseph Gil, editor, *proceedings of the* 26^{th} *conference on the Technology of Object-Oriented Languages and Systems*, pages 112–124. IEEE Computer Society Press, August 1998.

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

[95] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.* Prentice-Hall, 1^{st} edition, March 1979.

APPENDIX I

SOFTWARE METRICS

In our best knowledge, the following metrics defined by different organisation and different people. In the following, we present a list of all the metrics found during our research. For references and more information please check `ptidej.iro.umontreal.ca/Members/khosravk/metrics_home/`

1. ANMVC: *Average Number of Member Variables per Class* "The average amount of member variables or attributes of the classes within the software system".
2. AC: *Attribute Complexity metric* Defined by Chen and Lu.
3. Action: *Represents the number of actions the requirement needs to be capable of performing.*
4. ADI: *Average Depth of Inheritance* "Computed by dividing the sum of nesting levels of all classes by the number of classes".
5. AG: *Application Granularity:* "The number of objects per number of function points".
6. AHF: *Attribute Hiding Factor* Defined by Brito e Abreu as "the ratio of the sum of inherited attributes in all system classes under consideration to the total number of available classes attributes".
7. AIF: *Attribute Inheritance Factor* Defined by Brito e Abreu as "the ratio of the sum of inherited attributes in all classes of the system under consideration to the total number of available attributes for all classes".
8. AMC: *Average Method Complexity* "the sum of the cyclomatic complexity of all methods divided by the total number of methods".
9. AMC: *Average Method Complexity* "The average amount of complexity per method".
10. AML: *Average Module Length* "Traditional metrics Measures the average module size".
11. AMS: *Average Method Size* "The average size of program methods".
12. ANA: *Average Number of Ancestors* "The average number of ancestors of all the classes".
13. APG: *Application Granularity* "The total number of objects divided by the total number of function points".
14. ASC: *Association Complexity* "The complexity of the association structure of a system".
15. B: *Error estimate* "Estimated number of errors in the program".
16. BAM: *Binding Among Modules* "Traditional metrics Measures data sharing among modules".
17. BM1: Defined by Ian graham.
18. BM4 Defined by Ian graham.
19. Bounce-C: "Count of the number of yo-yo paths[1] visible to a CUT". Or it is defined as "a bounce may result in unanticipated binding". or "High bounce indicates more opportunities for faults".

[1] A yo-yo path is a path that traverses several supplier class hierarchies due to dynamic binding.

20. Bounce-S: “Count of the number-of yo-yo paths in SUT”. Or it is defined as “a bounce may result in unanticipated binding”. Or it is defined as “High bounce indicates more opportunities faults”.
21. Branch Count “This metric is the number of branches for each module. Branches are defined as those edges that exit from a decision node. The greater the number of branches in a program’s modules, the more testing resources required”.
22. Call Pairs: “Quantitative Metrics Number of calls to other functions in a module”.
23. CAN: *Category Naming* “Divides classes into semantically meaningful sets”.
24. CBO: *coupling between object classes* Defined by Chidamber and Kemerer (1994) as “counts the number of classes a class is coupled with”.
25. CCN: *Cyclomatic Complexity Number* “Traditional metrics Measures the number of decisions in the control graph”.
26. CCO: *Class Cohesion* “Measures relations between classes”.
27. CCP: *Class Coupling* “Measures connections between classes based on the messages they exchange”.
28. Cd: Defined by Brian Henderrson-sellers.
29. CDF: *Control flow complexity and Data Flow complexity* “Traditional metrics Combine metric based on variable definitions and cross-references”.
30. CEC: *Class Entropy Complexity* “Measures the complexity of classes based on their information content”.
31. CF: *Coupling Factor* “The ratio of the maximum possible number of couplings in the system to the actual number of couplings not imputable to inheritance”.
32. CFA: *Coupling Factor* “The ratio of the maximum possible number of couplings in the system to the actual number of couplings not imputable to inheritance”.
33. CH: *Class Hierarchy*
34. Classes: Defined by De Champeaux.
35. CLC: *Class Complexity* “The cyclomatic complexity of the control graph formed by a union of all method control graphs with a state transition graph for the class”.
36. ClCpl: *Class coupling metric* Defined by Chen and Lu.
37. CLM: *Comment Lines per Method* “Measures the percentage of comments in methods”.
38. COC: *Conditions and Operations Count* “Traditional metrics Counts pairs of all conditions and loops within the operations”.
39. COF: *Coupling Factor* Defined by Brito e Abreu.
40. Coh: *Cohesion Metric* Defined by Chen and Lu.
41. Condition Count: “Quantitative Metrics Number of conditionals in a given module”.
42. Conditional: *Represents whether the requirement will be addressing more than one condition. This indicates a higher level of complexity in dealing with multiple conditions within the requirement (i.e., If, when, . . .).*
43. COP: *Complexity Pair* “Traditional metrics combines cyclomatic complexity with logic structure”.

44. COR: *Coupling Relation* "Traditional metrics assigns a relation to every couple of modules according to the kind of coupling".
45. CRE: *Number of time a Class is Reused* "Measures the references in a class and the number of the applications that reuse this class".
46. CRM: *Cohesion Ratio Metrics* "Measure the number of modules having functional cohesion divided by the total number of modules".
47. D: *Program Difficulty* Defined by Halstead as "Level of difficulty in the program".
48. DAC: *Data Abstraction Coupling* Defined by Li and Henry as "the number of instances of ADTs or number of data member that having an ADT type".
49. DAM: *Data Access Metric* "The ratio of the number of private attributes to the total number of attributes declared in the class".
50. DCBO: *Degree of Coupling Between Objects:* "The average number of uses dependencies per object".
51. DCO: *Degree of Cohesion of Objects* "The degree of dependencies of parts within a single component".
52. dd: *Decision Density* Defined by McCabe software metrics.
53. DEC: *Decision Count* "Offers a method to measure program complexity".
54. Decision Count: "Number of decision points in a given module".
55. Dependency on Child: "(True/False) Whether a class is dependent on a descendant".
56. Depth: "The level for a class. For instance, if a parent has one child the depth for the child is two". Also defined as "depth indicates at what level a class is located within its class hierarchy. In general, inheritance increases when depth increases".
57. DIT: *Depth of inheritance Tree* Defined by Chidamber and Kemerer as "depths of inheritance of a class define as the depth in the inheritance tree".
58. DIT: *Depth of Inheritance Tree* "Measures the number of ancestors of a class".
59. DOC: *Dependency on Child (True/False)* "Whether a class is dependent on a descendant".
60. DRIM: *Degree of Reuse of Inheritance Methods* "The amount of methods that are actually reused in inheritance".
61. DSI: *Delivered Source Instructions* "Counts separate statements on the same physical line as distinct and ignore comment lines".
62. DYN: *Percent of Dynamic Calls* "Percent of messages throughout the SUT whose target is determined at runtime".
63. E: *Programming Effort* "Estimated mental effort required to develop the program".
64. Ed: *Essential Density* Defined by McCabe software metrics as "essential density is calculated as:$\frac{ev-1}{v-1}$".
65. Edge Count: "Number of edges found in a given module. Represents the transfer of control from one module to another. (Edges are a base metric, used to calculate many of the more involved complexity metric)".
66. ERE: *Extent of Reuse* "Categorises a unit according the lever of reuse: modifications required".

67. Error Density: *Error density* "Error Metrics ER/KLOC".
68. Error Rate: *Error rate* "The number of changes due to error. If a module is changed due to an error report (as opposed to a change request), then it receives a one up count. It cannot receive more than a one up for a given error report".
69. ESM: *Equivalent Size Measure* "Measures the percentage of modifications on a reused module".
70. EST: *Executable Statements* "Counts separate statements on the same physical line as distinct and ignore comment lines, data declarations and headings".
71. Ev: *Essential Complexity* Defined by McCabe software metrics as "essential complexity is a measure of the degree to which a module contains unstructured constructs".
72. Fan-in: "Count of calls by higher modules".
73. FCO: *Function Count* "The number of functions and the source lines in every function".
74. FDE: *Functional Density* "The ratio of LOC to the function points".
75. FEF: *Factoring Effectiveness* "The number of unique methods divided by the total number of methods".
76. FIN: *FAN-IN* "The number of classes from which a class is derived and high values indicate excessive use of multiple inheritance".
77. FOC: *Function Oriented Code* "Measures the percentage of non objectoriented code that is used in a program".
78. Formal Parameter Count: "Number of parameters to a given module".
79. FUP: *Function Points* "Measures the amount of functionality in a system".
80. gd: *Global Data Density* Defined by McCabe software metrics as "global Data density is calculated as: $\frac{gdv}{v}$".
81. gdv: *Global Data Complexity* Defined by McCabe software metrics as "global Data Complexity quantifies the cyclomatic complexity of a module's structure as it relates to global or parameter data".
82. GLM: *Global Modularity* "Describes global modularity in terms of several specific views of modularity".
83. HNL: *Class Hierarchy Nesting Level* "Measures the depth in hierarchy that every class is located".
84. i: *Intelligent Content* Defined by Halstead as "complexity of a given algorithm independent of the language used to express the algorithm".
85. IC: *Inheritance Complexity* Defined by Moreau and Dominck as " Compound and multilevel inheritance contribute to the complexity of building, understanding, and maintainable object. They believe that size of the inheritance tree is a simple approximation to such a metric".
86. Id: *Design Density* Defined by McCabe software metrics as "design density is calculated as: $\frac{iv(G)}{v(G)}$".
87. ID: *Inheritance Dependencies* "The depth of inheritance".

88. IFL: *Information Flow* "Measures the total level of information flow between individual modules and the rest of a system".
89. IL: *interaction level* Defined by Abbott, Korson and McGregor as "the degree of interaction between two objects".
90. Incomplete: "Phrases such as TBD or TBR. They are used when a requirement has yet to be determined. These are considered critical to requirements documents and need to be corrected as soon as possible. They can cause unexpected delays and high costs".
91. INP *Internal Privacy:* "Refers to the use of accessory functions even within a class".
92. Iv: *Design Complexity*
93. Imperative: Requirement Metrics McCabe software metrics as "design complexity is a measure of a module's decision structure as it relates to calls to other modules. This quantifies the testing effort related to integration".
94. KNM: *Knot Measure* "Traditional metrics the total number of crossing points on control flow lines".
95. L: *Program Level* Defined by Halstead Metrics as "level at which the program can be understood".
96. LCM: *Lack of Cohesion between Methods* "Indicates the level of cohesion between the methods".
97. LCOM: *Lack of Cohesion in methods* Defined by Chidamber and Kemerer (1993): "LCOM is a count of the number of method pairs whose similarity is zero, minus the count of method pairs whose similarity is not zero. where similarity of a pair of methods is the number of joint instance variables4 used by both methods".
98. LCOM: *Lack of Cohesion in Methods* Defined by Henderson-Sellers (1995).
99. Length of the variable names: *Length of the variable names* "Measuring the length of each of the Variable names, and by doing so we can probably tell whether that variable is more or less understandable then the others".
100. LOC: *Lines Of Code* "Measures the size of a module".
101. LVA: *Live Variables* "Traditional metrics deals with the period each variable is used".
102. MAA: *Measure of Attribute Abstraction* "The ratio of the number of attributes inherited by a class to the total number of attributes in the class".
103. MAG: *MAX V(G)* "The maximum cyclomatic complexity of the methods of one class".
104. MCX: *Method Complexity* "Relates complexity with the number of messages".
105. MDS: *Message Domain Size* Defined by Moreau and Dominck as "the number of distinct procedures within the object that manipulate its state, thus the number of types of messages to which an object will respond".
106. MFA: *Measure of Functional Abstraction* "The ratio of the number of methods inherited by a class to the total number of methods accessible by members in the class".
107. MHF: *Method Hiding Factor* Defined by Brito e Abreu as "ObjectOriented CLASS METRICS defined as the ratio of the sum of the invisibilities of all methods defined in all classes to the total number of methods defined in the system under consideration".

108. MIF: *Method inheritance factor* Defined by Brito e Abreu as "ObjectOriented INHERITANCE METRICS the ratio of the sum of the inherited methods in all classes to the total number of available methods for all classes".

109. MNP: *Minimum Number of Paths* "Measures the minimum number of paths in a program and the reachability of any node".

110. Modified Condition Count: "Every condition shown to independently affect a decision outcome (by varying that condition only)".

111. MOR *Morphology metrics:* "Measure morphological characteristics of a module, such as size, depth, width and edge-to-node ratio".

112. MPC: *Message Passing Coupling* Defined by Li and Henry as "Coupling Metrics Defined as the number message that sent out from a method in a class to a method in another class".

113. MPC: *Message-Passing Coupling* Proposed separately by Li & Henry(1993), and Lorenz & Kidd (1994)Coupling Metrics

114. MPC: *Methods Per Class* "Average number of methods per object class".

115. MRE: *Method Reuse metrics* "Indicate the level of methods reuse".

116. Multiple Condition Count: "Quantitative Metrics Number of multiple conditions that exist within a module".

117. MVS: *Message Vocabulary Size* Defined by Moreau and Dominck as "The number of diferent types of message sent by a particular object; it is related to the number of functions that the programmer must be familiar with to comprehend the object".

118. N: *Program Length* Defined by Halstead Metrics as "This is a Halstead metric that includes the total number of operator occurrences and total number of operand occurrences".

119. Np: *Number of Progeny* "Class level Complexity Metric The number of subclasses that inherit directly or indirectly from a class".

120. NAD: *Number of Abstract Data types* "The number of user-defined objects used as attributes in a class that are necessary to instantiate an object instance of the class".

121. NCM: *Number of Class Methods in a class* "Measures the measures available in a class but not in its instances".

122. NCT: *Number of Classes Thrown away* "Measures the number of times a class is rejected until it is finally accepted".

123. NIV: *Number of Instance Variables in a class* "Measures relations of a class with other objects of the program".

124. NLE: *Nesting Levels* "Traditional metrics measures the complexity as depth of nesting".

125. NMI: *Number of Methods Inherited* "Measures the number of methods a class inherits".

126. NMO: *Number of Methods Overridden* "The number of methods needs to be re-declared by the inheriting class".

127. NOA: *Number Of Ancestors* Defined by De Champeaux as "ObjectOriented CLASS METRICS the total number of ancestors of a class".

128. NOAM: *Number Of Accessor Methods* "The number of the non-inherited accessor methods declared in the interface of a class. Interpretation: If a class has a high NOAM value a part of the functionality of that class is probably misplaced in one or more other classes".
129. NOC: *Number of Children* "The number of classes derived from a specified class". Or defined as "number of immediate subclasses subordinated to a class in the class hierarchy".
130. Node Count: "Number of nodes found in a given module. (Nodes are a base metric, used to calculate many of the more involved complexity metrics)".
131. NOH: *Number Of Hierarchies* "The number of distinct hierarchies of the system".
132. NOM: *The number of methods* Defined by Li and Henry as "number of local methods that defined as two sizes": size 1, The number of semicolons (In class defined as LOC metrics); size 2, The number of properties(the number of attributes + the number of local methods).
133. NOPA: *Number Of Public Attributes* "The number of non-inherited attributes that belong to the interface of a class. Interpretation: Classes with public data members violate encapsulation and couples its clients to its structure".
134. NOR: *Number of Root Classes* "The number of different class hierarchies in the system".
135. Norm v: *Normalized Cyclomatic Complexity* Defined by McCabe software metrics as "Normalized Cyclomatic Complexity is calculated as: $\frac{v}{nl}$".
136. NPA: *Number of Public Attributes* "Counts the number of attributes declared as public in a class".
137. NPM: *Number of Parameters per Method* "The average number of parameters per method in a class".
138. NRA: *Number of Reference Attributes* "Counts the number of pointers and references used as attributes in a class".
139. OAC: *Operation argument complexity* Defined by Chen and Lu.
140. OLE: *Object Library Effectiveness* "The ratio of the total number of object reuses divided by the total number of library objects".
141. OpCom: *Operation Complexity* Defined by Chen and Lu as "Complexity of operation in each method".
142. OpCpl: *Operation coupling metric* Defined by Chen and Lu as "Coupling percentage in each classes".
143. OVR: *Percent of Non-Overloaded Calls* "Percent of calls throughout the SUT that are not made to overload modules".
144. PAD: *Public Access to Data Members* "The number of external accesses to a classs public or protected data members".
145. PAP: *Percentage Public and Protected* "The percentage of public member variables in the related class".
146. PCM: *Percentage of Commented Methods* "The percentage of commented methods".
147. PDA: *Public Data* "Counts the accesses of public and protected data of a class".
148. PF: *Polymorphism Factor* "The amount of redefined inherited methods in the system."

149. PFA: *Polymorphism Factor* "The ratio of the actual number of possible different polymorphic situations of a class to the maximum number of possible distinct polymorphic situations for this class".

150. PMO: *Percent of Potential Method uses Overridden* "The percentage of the overridden methods".

151. PMR: *Percent of Potential Method uses actually Reused* "The percentage of the actual method uses".

152. PPD: *Percentage of Public Data* "The percentage of the public data of a class".

153. PRC: *Problem Reports per Class* "Measures defect reports on this class".

154. PRO: *Percent of Reused Objects Modified* "The percentage of the reused objects that have been modified".

155. PRs in 1 Yr.: *Error Reports in One Year*

156. PRs in 2 Yrs: *Error Reports in Two Years*

157. PRs in 6 Mths: *Error Reports in Six Months*

158. Pub Data: "The amount of times that a class's public and protected data is accessed. In general, lower values indicate greater encapsulation. It is a measure of encapsulation".

159. Pv: *Pathological Complexity* Defined by McCabe software metrics as "A measure of the degree to which a module contains extremely unstructured constructs".

160. RDB: *Ratio between Depth and Breadth* "The ratio between the depth and the width of the hierarchy of the classes".

161. Relevance of classes: *Relevance of classes*

162. Relevance of methods: *Relevance of methods* "Measuring the importance (usage) of a single method in the class compared to other methods. It is possible to say that if a method is used more then other methods in the program then it is more relevant to the program".

163. Relevance of variables: *Relevance of variables* "Measuring the importance (usage) of a single variable compared to other variables in the same class ,if a class changes the value of a variable more then the other variables in the class , it is possible to say that this variable is more relevant to the program".

164. RER: *Reuse Ratio* "The ratio of the number of superclasses divided by the total number of classes".

165. RFC: *Response for Class* Defined by Chidamber and Kemerer as "ObjectOriented CLASS METRICS A count of methods implemented within a class plus the number of methods accessible to an object class due to inheritance. In general, lower values indicate greater polymorphism".

166. SIX: *Specialisation Index* "Measures the type of specialization".

167. Source: "Represents the number of sources the requirement will interface with or receive data from".

168. SPR: *Specialisation Ratio* "The ratio of the number of subclasses divided by the number of superclasses".

169. SRE: *System Reuse* "Declares the percentage of the reuse of classes".

170. SSC: *composite metric of Software Science and Cyclomatic complexity* "Combines software science metrics with McCabe's complexity measure".

171. SWM: *Specification Weight Metrics* "Measure the function primitives on a given data flow diagram".

172. T: *Programming time* "Estimated amount of time to implement the algorithm".

173. The efficacy of the subclasses: *The efficacy of the subclasses* "Measuring the ratio of inheritance from the super-class into the Subclasses".

174. The size of the main class: *The size of the main class* "Measuring the number of lines in the main class in relation to the total program size in order to see whether the program is modular".

175. TOF: *Total Number of Functions per Class* "The larger number of functions, the larger amount of testing needed./ Search the abstract tree from the root node down and total up the functions".

176. TOM: *Total Number of Methods per Class* "The total number of methods in a class, which includes all inherited methods./ Search the abstract tree from the root node down and total up the methods". Or defined as "the larger number of methods, the larger amount of testing needed".

177. TOP: *Total Number of Procedures per Class* "The total number of procedures in a given class".

178. Total Operands: "Variables and identifiers Constants (numeric literal/string) Function names when used during calls".

179. TRI: *Tree Impurity* "Determines how far a graph deviates from being a tree".

180. TRU: *Transfer Usage* "The logical structure of the program".

181. Unique Operands "Variables and identifiers Constants (numeric literal/string) Function names when used during calls".

182. V: *Cyclomatic Complexity* Defined by McCabe software metrics as "It is a measure of the complexity of a modules decision structure. It is the number of linearly independent paths".

183. V: *Program Volume* Defined by Halstead Metrics as "the minimum number of bits required for coding the program".

184. vd: *Cyclomatic Density* Defined by McCabe software metrics as "the ratio of the module's cyclomatic complexity to its length in NCSLOC. The intent is to factor out the size component of complexity. It has the effect of normalizing the complexity of a module, and therefore its maintenance difficulty".

185. VOD: *Violations of Demeter* Defined by Haynes and Menzies(1994) and Equation & Henderson-Sellers (1995) as "Coupling Metrics Number of times the Law of Demeter is violated".

186. WCS: *Weighted Class Size* "Weighted Class Size is the number of ancestors plus the total class method size".

187. WMC: *weighted methods per class* Defined by Chidamber and Kemerer as "ObjectOriented CLASS METRICS the sum of the weights of all the class methods".

188. WMC: *weighted methods per class* Defined by Henderson Sellers as "A count of methods implemented within a class (rather than all methods accessible within the class hierarchy). In general, lower values indicate greater polymorphism".

189. WMC: *Weighted Methods per Class* Defined by Chidamber & Kemerer's as "The measure of the complexity of a single class; the weighting is related to the complexity of the class as measured by the number of methods, m, and the complexity of each, V_G(m)".

APPENDIX II

QUALITY CHARACTERISTICS

Many things difficult ro design prove easy to performance.
–Samuel Johnson

Definitions of quality characteristic have direct relations with the programming language and the environment for which a software product is implemented. For example, Lowell J. Arthur in 1951 defines the flexibility quality characteristic using the question: "Is the program free of spaghetti code?" [5], *i.e.,* does the program source code contains GOTO instructions? Thus, this definition of the flexibility quality characteristic relates to pre-procedural structural programming directly and is no longer practical for object-oriented programs.

Also for better understanding the concept of quality to better implementation of our idea, we have to know enough about the definition of software characteristics and sub-characteristics which defined in literature.

II.1 Definitions

In the following, we summarise standard or latest definitions for quality characteristics related to object-oriented programs and used in other sections of this thesis to define the quality models. These definitions are sorted alphabetically.

1. Accessibility "Accessibility is the degree to which the user interface of something enables users with common or specified (e.g., auditory, visual, physical, or cognitive) disabilities to perform their specified tasks" [32]. "Does the model facilitate selective use of its parts for other purposes (e.g., for the construction of another model)?" [6].

2. Accountability "Accountability: Does the model lend itself to measurement of its usage? Can probes be inserted to measure timing, whether specified branches are exercised, etc.?" [6].

3. Accuracy "The capability of the software product to provide the right or agreed results or effects with the needed degree of precision" [48]. Also, "[t]he precision of computations and control" [72], the "[a]ttributes of software that bear on the provision of right or agreed results or effects" [78], "the magnitude of defects (*i.e.,* the deviation of the actual or average measurements from their true value) in quantitative data" [32]. "Are l;he models calculations and outputs sufficiently precise to satisfy their intended use?" [6].

4. Adaptability "The capability of the software product to be adapted for different specified environments without applying actions or means other than those provided for this purpose for [/by] the software considered" [48]. Also, "[a]ttributes of software that bear on the opportunity for its adaptation to different specified environments without applying other actions or means than those provided for this purpose for the software considered" [78].

   Adaptability mostly considering as following options:

   - Independence of storage: We need to ensure that software modules are independent of storage size to make the software more adaptable.
   - Uncommitted memory: the ability to allocate address space without allocating memory to back it up at the same time

- Uncommitted processing capacity: is defined as percentage of uncommitted processing capacity.

5. Adaptivity "Adaptivity suggests that the system should be designed to the needs of different types of users" [3].
6. Ambiguity This characteristic relates with requirements with potential multiple meanings [44].
7. Analyzability "The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified" [48]. Also, the "[a]ttributes of software that bear on the effort needed for diagnosis of deficiencies or causes of failures, or for identification of part to be modified" [78].
8. Attractiveness "The capability of the software product to be attractive to the user" [48]. "Attractiveness is achieved through layout, graphics, color, and dynamic elements" [3].
9. Auditability "The ease with which conformance to standards can be checked" [72].
10. Augmentability The ability of "the model to accommodates expansion in component computational functions or data storage requirements" [6]. This attribute relates to support the growth of data storage.
11. Availability "Availability is the degree to which a work product is operational and available for use" [32] as a product or to uses. Availability has the same definition for malicious and non-malicious users.
12. Behavior

    - Time behavior: "The capability of the software product to provide appropriate response and processing times and throughput rates when performing its function" [48].
    - Resource behavior: The attributes of software related with measuring the amount of resources required to perform its function [48].

13. Branding "Branding is the degree to which a work product (e.g., application, component, or document) successfully incorporates the brand of the customer organization's business enterprize" [32].
14. Capacity "Capacity is the minimum number of things (e.g., transactions, storage) that can be successfully handled." [32].
15. Configurability "Configurability is the degree to which something can be configured into multiple forms (*i.e.,* configurations)" [32].
16. Changeability "The capability of the software product to enable a specified modification to be implemented" [48]. Also, the "[a]ttributes of software that bear on the effort needed for modification, fault removal or for environmental change" [78]. Changeability is also called "modifiability" [3].
17. Co-existence "The capability of the software product to co-exist with other independent software in a common environment sharing common resources" [48].
18. Compatibility "Compatibility is the degree to which a system or a component can be used and functions correctly under specified conditions of the physical environment(s) in which it is intended to operate" [32].

19. Completeness "The degree to which full implementation of required function has been achieved" [72]. Also, completeness relates to requirements, documentation, and comments for explain the program input and their presence with comments, without reference to dummy programs.

20. Compliance "Attributes of software that make the software adhere to application-related standards of conventions or regulations in laws and similar prescriptions" [78]. Also, degree to which the software is found to "[c]omply with relevant standards and practices" [3]. In [48], compliance decomposes in:

    - Portability compliance: "The capability of the software product to adhere to standards or conventions relating to portability".
    - Maintainability compliance: "The capability of the software product to adhere to standards or conventions relating to maintainability".
    - Efficiency compliance: "The capability of the software product to adhere to standards or conventions relating to efficiency".
    - Usability compliance: "The capability of the software product to adhere to standards, conventions, style guides or regulations relating to usability".
    - Reliability compliance: "The capability of the software product to adhere to standards, conventions or regulations relating to reliability".
    - Functionality compliance: "The capability of the software product to adhere to standards, conventions or regulations in laws and similar prescriptions relating to functionality".

21. Communication commonality "The degree to which standard interfaces, protocols and bandwidths are used" [72].

22. Communicativeness "Does the model facilitate the specification of inputs? Does it provide outputs whose form and contenl, are easy to assimilate and useful?" [6]

23. Computability This attribute relates to computation safety (such as division by zero or other impossible computations).

24. Completeness "Are all model inputs used within the model? Are there no dummy sub-models referenced?" [6]

25. Conformance "Attributes of software that make the software adhere to standards or conventions relating to portability" [78].

26. Conciseness "The compactness of the program in terms of lines of code" [72]. Also, "Attributes of software that provide the implementation of a function with minimum amount of code" [85]. Conciseness relates to program excess, for example, unused entities (types, objects, parameters) or internal invocations of other functions within the same file decrease the value of conciseness. Conciseness answer the following questions: "Is the model implemented with a minimum amount of code? Is it excessively fragmented into sub-models so that the same sequence of code is not repeated in numerous places?" [6].

27. Consistency This attribute answer the following questions [6]:

    - Does the model contain uniform notation, terminology, and symbology within itself?
    - Are all model attributes and variables typed and specified consistently for all uses?
    - Are coding standards homogeneously adhered to?"

28. Configurability "The ability to organize and control elements of the software configuration" [72].
29. Consistency "The use of uniform design and documentation techniques throughout the software development project" [72].

    For example:

    - The set of global variables is supposed to be used across more than one sub-program.
    - The type of variables is supposed to be consistent for all their uses.
30. Correctability "Correctability is the ease with which minor defects can be corrected between major releases while the application or component is in use by its users" [32].
31. Correctness The "[e]xtent to which a program satisfies its specifications and fulfills the user's mission objectives" [36, 72]. "Correctness is the degree to which a work product and its outputs are free from defects once the work product is delivered" [32]. Correctness answers the following typical questions: "Is the application and its data complete, accurate and consistent?" [5].
32. Currency "Currency is the degree to which data remain current (*i.e.,* up to date, not obsolete)" [32].
33. Data Commonality "The use of standard data structures and types throughout the program" [72].
34. Dependability "Dependability is the degree to which various kinds of users can depend on a work product" [32].
35. Device independability

    - Factors for independency between computations and the computer configuration.
    - Factors for independency between computations and hardware capability, half word accessing, bit patterns...

    Also, this attribute answer the of following questions [6]:

    - Can the model be executed on other computer hardware configurations?
    - Have machine-dependent statements been flagged and documented?"
36. Effectiveness "The capability of the software product to enable users to achieve specified goals with accuracy and completeness in a specified context of use" [48].
37. Efficiency "The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions" [48]. "Efficiency is the degree to which something effectively uses (*i.e.,* minimizes its consumption of) its resources. These resources may include all types of resources such as computing (hardware, software, and network), machinery, facilities, and personnel" [32]. Also, "[t]he amount of computing resources and code required by a program to perform a function" [36,72], "[a] set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used under stated conditions " [78]. Efficiency relates to "shed load, end-to-end error detection: Cheap test, Performance defects appear under heavy load, safety first, scaling, throughput, latency, availability" [3]. Also with efficiency we looking for answer of this question:"Does the model fulfill its objective without waste of resources?" [6].
38. Error tolerance "The damage that occurs when the program encounters an error" [72].

39. Expendability "The degree to which architectural, data or procedural design can be extended" [72].
40. Extendibility The attributes related to the modification of a component or a system in case of increase of the storage or of the functional capacity [77].
41. Extensibility "Extensibility is the ease with which an application or component can be enhanced in the future to meet changing requirements or goals" [32]. Also, attributes related to new capabilities or to the modification of existing capabilities upon user needs [77].
42. Fault Tolerance "The capability of the software product to maintain a specified level of performance in cases of software faults or of infringement of its specified interface" [48]. Also, the "[a]ttributes of software that bear on its ability to maintain a specified level of performance in cases of software faults or of infringement of its specified interface" [78]. ("Use [of] robust methods to protect against permanent failure of a limited number of components. Use [of] stabilizing methods to protect against transitory faults" [3].)
43. Flexibility "Effort required to modify an operational program" [36]. The effort to change or to modify a software product to adapt it to other environment or to other applications different from which it was designed.
44. Functionality "The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions" [48]. Functionality is "[a] set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs" [78]. Functionality "[i]s assessed by evaluating the feature set and capabilities of the program, the generality of functions that are delivered and the security of overall system" [72].
45. Generality "The breadth of potential application of program components" [72]. Generality is defined as the degree to which a software product can perform a wide range of functions.
46. Hardware independence "The degree to which the software is decoupled from the hardware on which it operates" [72].
47. Independence of storage The ability to bring new storage where needed at a moments notice, more resilience and automatic failure recovery, enhanced performance, and cost savings from efficient storage use [37].
48. Indicesability

    Indicesability is defined as a system of numbers used for comparing values of things which vary against each other or against a fixed standard. This attributes relates to the degree of correctness of the software product throughout its development cycle.
49. Initializability This attribute relates to the degree a software product can be initialized with the expected values.
50. Installability "The capability of the software product to be installed in specified environment" [48]. "Installability is the ease with which something can be successfully installed in its production environment(s)" [32]. Also, the "[a]ttributes of software that bear on the effort needed to install the software in a specified environment" [78].
51. Instrumentation "The degree to which the program monitors its own operation and identifies errors that do occur" [72].
52. Integrity The "[e]xtent to which access to software or data by unauthorized persons can be controlled" [36, 72]. Also, the attributes related to control a software product for illegal accesses to the program and its data [32].

53. Interface facility The degree to which two software products can be connected successfully.

54. Internationalization "Internationalization (also known as globalization and localization) is the degree to which something can be or is appropriately configured for use in a global environment" [32].

55. Interoperability "The capability of the software product to interact with one or more specified systems" [48]. Also, the "[e]ffort required to couple one system with another" [36, 72], the "[a]ttributes of software that bear on its ability to interact with specified systems" [78], "the degree to which a system or one of its components is properly connected to and operates with something else" [32].

56. Learnability "The capability of the software product to enable the user to learn its application" [48]. Also, the "[a]ttributes of software that bear on the users' effort for learning its application" [78]. "Learnability requires attention to the needs of the novice and uninitiated users. The uninitiated user is one that has no previous experience with the software or similar software. The novice user has either had some experience with similar software or has limited experience with the software" [3].

57. Legibility This attribue answer the following question: "Does the model possess the characteristic that its function is easily discerned by reading the code?" [6].

58. Maintainability "The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to change in environment, and in requirements and functional specifications" [48]. Also, the "[e]ffort required to locate and fix an error in an operational program" [36,72]. "Maintainability is the ease with which an application or component can be maintained between major releases" [32]. Also, "[a] set of attributes that bear on the effort needed to make specified modifications" [78], the degree of changing or modifying the components to correct errors, to improve performance, or to adapt for changing the environment.

59. Maturity "The capability of the software product to avoid failure as a result in the software" [48]. Also, the "[a]ttributes of software that bear on the frequency of failure by faults in the software" [78].

60. Modularity "The functional independence of program components" [72]. Modularity is increased when it is possible to divide each components into sub-components.

61. Operability "The capability of the software product to enable the user to operate and control it" [48]. Also, "[t]he ease of operation of a program" [72]. "Operability is the degree to which something enables its operators to perform their tasks in accordance with the operations manual" [32]. Also, the "[a]ttributes of software that bear on the users' effort for operation and operation control" [78]. "Part of the design process for operability is to develop scenarios and use cases for novice, uninitiated, and expert users. Operability is enhanced through navigational efficiency, *i.e.,* users can locate the information they want" [3].

62. Performance "Performance is the degree to which timing characteristics are adequate" [32]. Performance "[i]s measured by evaluating processing speed, response time, resource consumption, throughput, and efficiency" [72].

63. Personalization "Personalization is the degree to which each individual user can be presented with a unique user-specific experience" [32].

64. Portability "The capability of the software product to be transferred from one environment to another" [48]. Also, the "[e]ffort required to transfer a program from one hardware configuration and–or software system environment to another" [36, 72]. "Portability is the ease with which an application or component can be moved from one environment to another" [32, 78].

65. Precision "Precision is the dispersion of quantitative data, regardless of its accuracy" [32].

66. Productivity "The capability of the software product to enable users to expand appropriate amounts of resources in relation to the effectiveness achieved in specified context of use" [48].

67. Readability "Readability is characterized by clear, concise code that is immediately understandable" [5]. Readability is defined as the set of attributes related to the difficulty in understanding software components source and documentation.

68. Recoverability "The capability of the software product to re-establish a specified level of performance and recover the data directly affected in the case of failure" [48, 78]. Recoverability "[u]se[s] recovery oriented methods. System architecture should be designed with components that can be restarted independently of the other components. System architecture should be designed with an undo function to rewind time, untangle problems, and replay the system back to the current time" [3].

69. Reliability "The capability of the software product to maintain a specified level of performance when used under specified conditions" [48]. Reliability is the "Extend to which a program can be expected to perform its intended function with required precision" [36,72]. It "[i]s evaluated by measuring the frequency and severity of failure, the accuracy of output result, the mean time between failure (MTBF), the ability to recover from failure and the predictability of the program" [72] because "Unreliable programs fail frequently, or produce incorrect data" [5]. Also, reliability is "[a] set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for stated period of time" [78]. "Reliability is the degree to which a work product operates without failure under given conditions during a given time period" [32].

70. Replaceability "The capability of the software product to be used in place of another specified software product for the same purpose in the same environment" [48]. Also, "[a]ttributes of software that bear on opportunity and effort of using it in place of specified other software in the environment of software" [78].

71. Requirements Risk Attributes related to the risk of project failure because of requirements (as with poorly written or rapidly changing requirement).

72. Responsiveness "Responsiveness is the ability of a system to meet its objectives for response time or throughput. In end-user systems, responsiveness is typically defined from a user perspective" [82].

73. Resource utilization "The capability of the software product to use appropriate amounts and types of resources when the software performs its function under stated conditions" [48].

74. Reusability "Reusability is the ease with which an existing application or component can be reused" [32]. It is the "[e]xtent to which a program can be used in other applications related to the packaging and scope of the functions that programs perform" [36, 72]. For example, reusability is possible when "[m]any modules contain two or more unique functions which, if separated from the main body of code, could be reused with other programs" [5]. Also, he attributes related to the cost of transferring a module or program to another application.

75. Robustness "Robustness is the degree to which an executable work product continues to function properly under abnormal conditions or circumstances" [32]. Also, the attributes related to the correct functioning of a software product in the case of invalid inputs or under stressful environmental conditions. "Does the model continue to execute reasonably when it is run with invalid inputs? Can the model assign default values to non-specified input variables and parameters? Does the model have the capability to check input data for domain errors?" [6]

76. Safety "The capability of the software product to achieve acceptable levels of risk of harm to people business, software, property or the environment in specified context of use" [48].

77. Satisfaction "The capability of the software product to satisfy users in specified context of use" [48].

78. Scalability "Scalability is the ease with which an application or component can be modified to expand its existing capacities" [32]. "[S]calability is crucial for keeping costs down and minimizing interruptions in production" [31]. "Scalability is the ability of a system to continue to meet its response time or throughput objectives as the demand for the software functions increases" [82].

79. Scheduleability "Scheduleability is the degree to which events and behaviors can be scheduled and then occur at their scheduled times" [32].

80. Security "The capability of the software product to protect information and data so that unauthorized persons or systems cannot read or modify them and authorized persons or systems are not denied access to them" [48]. Also, security is "[t]he availability of mechanisms that control of protect programs and data" [72], "[a]ttributes of software that bear on its ability to prevent unauthorized access, where accidental or deliberate, to programs and data" [78].

81. Self containedness "Self containedness is related to the facility of the software product for initializing core storage prior to use and for proper positioning of input/output devices prior to use" [6].

82. Self-descriptiveness This attribute answer the following question: "Does the model contain enough information for a reader to determine or verify its objectives, assumptions, constraints, inputs, out,puts, components, and revision status?" [6].

83. Self documentation "The degree to which the source code provides meaningful documentation" [72].

84. Simplicity "The degree to which a program can be understood without difficulty" [72].

85. Software system independence "The degree to which the program is independent of nonstandard programming language features, operating system characteristics, and other environmental constraints" [72].

86. Stability "The capability of the software product to avoid unexpected effects from modifications of the software" [48]. Also, the "[a]ttributes of software that bear on the risk of unexpected effect of modifications" [78].

87. Structuredness This attribute answer the following question: "Does the model possess a definite pattern of organization of its interdependent parts?" [6].

88. Subsetability "Subsetability is the degree to which something can be released in multiple variants, each of which implements a different subset of the functional requirements and associated quality requirements" [32].

89. Suitability "The capability of the software product to provide an appropriate set of functions for specified tasks and user objectives" [48]. Also, the "[a]ttributes of software that bears on the presence and appropriateness of a set of functions for specified tasks" [78].

90. Supportability Supportability "combines the ability to extend the program (extensibility), adaptability and serviceability (these three attributes represent a more common term—maintainability), in addition to testability, computability, configurability, the ease with which a system can be installed and the ease with which problems can be localized" [72].

91. Survivability "Survivability is the degree to which essential, mission-critical services continue to be provided in spite of either accidental or malicious harm" [32].

92. Testability "The capability of the software product to enable modified software to be validated" [48]. Also, the "[e]ffort required to test a program to insure it performs its intended function" [36, 72]. "Testability is the ease with which an application or component facilitates the creation and execution of successful tests (*i.e.,* tests that would cause failures due to any underlying defects)" [32]. Also, the "[a]ttributes of software that bear on the effort needed for validating the modified software" [78].

93. Traceability "The ability to trace a design representation or actual program component back to requirements" [72]. Traceability is defined as the attributes that increase traceability among implementation, design, architecture, requirements...

94. Training "The degree to which the software assists in enabling new users to apply the system" [72].

95. Transferability This attribute relates to the cost of transferring a software product from its original hardware or operational environment to another.

96. Transportability "Transportability is the ease with which something can be physically moved from one location to another" [32].

97. Trustability "Trustability refers to the system's ability to provide users with information about service correctness" [3].

98. Uncommitted memory "Uncommitted memory enables an application to differentiate between reserving and using (committing) address space" [70]

99. Uncommitted processing "Capacity is amounts of unattached processing capacity [27].

100. Understandability "The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use" [48]. Also, the "[a]ttributes of software that bear on the users' effort for recognizing the logical concept and its applicability" [78].

101. Usability "The capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions" [48]. Usability is related to the "set of attributes that bear on the effort needed for use, and on the individual evaluation of such use, by a stated or implied set of users" [10, 78]. Also, usability is the "[e]ffort required to learn, operate, prepare input, and interpret output of program" [36, 72], "the ease with which members of a specified set of users are able to use something effectively" [32]. Usability "[i]s assessed by considering human factors, overall aesthetics, consistency, and documentation" [72].

102. Utility "Utility is the degree to which something can be accessed and used by its various types of users" [32].

103. Variability "Variability is the degree to which something exists in multiple variants, each having the appropriate capabilities" [32].
104. Verifiability "Verifiability is the ease with which an application or component can be verified to meet its associated requirements and standards" [32].
105. Volatility This attribute relates to the requirements documents when the software product changes frequently.
106. Withdrawability "Withdrawability is the ease with which an existing problematic version of the system or one of its components can be successfully withdrawn and replaced by a previously working version" [32].

II.2 Characteristics and Products

Table II.1, shows the software characteristics, which we found references in the literature in relation to different software products.

Quality Characteristics	All Products	Requirements Documentation	Design Documentation	Code	Test Documentation
Completeness	X [77]	X [77]	X [77]	X [77]	X [77]
Correctness	X [77]	X [77]	X [5]	X [77]	X [5]
Reliability	X [77]		X [5]	X [77]	
Generality		X [77]	X [5]	X [77]	
Understandability		X [77]	X [5]	X [77]	X [77]
Efficiency			X [5]	X [77]	X [77]
Modularity			X [5]	X [77]	
Portability			X [5]	X [77]	
Reusability				X [5]	
Adaptability				X [77]	
Maintainability				X [77]	
Flexibility				X [5]	
Usability				X [5]	
Testability				X [5]	
Integrity				X [5]	
Interoperability				X [5]	

Table II.1: Interrelationships among software characteristics and software products

APPENDIX III

INTERRELATIONS AMONG QUALITY CHARACTERISTICS

Some quality characteristics are related to one another, we summarise these interrelationships among software quality characteristics.

1. Adaptability: Direct related with Expandability.
2. Completability: Direct related with Functionality, Reliability, Usability, and Maintainability.
3. Computability: Direct related with Functionality and Reliability.
4. Correctness: Direct related with Reliability [5, 33, 63], Usability [5, 33, 63], Maintainability [5, 33, 63], Testability [5, 33, 63] and Flexibility [5, 33, 63].
5. Efficiency: Direct related with Field Performance [5].
   Inverse related with Execution Efficiency [63], Software system independence [63], Integrity [5, 33, 63], Usability [5, 33, 63], Maintainability [5, 33, 63], Testability [5, 33, 63], Flexibility [5, 33, 63], Portability [5, 33, 63], Reusability [5, 33, 63], Completeness [63], Consistency [63], Traceability [63], and Interoperability [5, 33, 63].
6. Expandability: Direct related with Augmentability, Extendability and Extensibility.
7. Flexibility: Direct related with Computablability [28], Completeness [28], Expandability [63], Generality [63], Modularity [63], Self Documentation [63], Correctness, Reliability, Usability, Maintainability and Testability.
   Inverse related with Performance [5], Reusability [5, 33, 63] and Efficiency.
8. Installability: Direct related with Availability.
9. Integrity: Direct related with Access audit [63], Access control [63], Usability [5, 33, 63].
   Inverse related with Flexibility [5, 63], Reusability [5, 33, 63], Interoperability [5, 33, 63], and Efficiency.
10. Interoperability: Direct related with Performance [5], Enhancement Costs (Selection Criteria) [5], Communication Communality [63], Data Commonality [63], Modularity [63], and Portability.
    Inverse related with Efficiency and Integrity.
11. Mainainability: Direct Related with Completeness [28], Structuredness [28], Effectiveness [28], Conciseness [63], Consistency [63], Modularity [63], Self documentation [63], Simplicity [63], Correctness [74], Testability [33, 63, 74], Modifiability [74], Flexibility [63], Portability [63], Reusability [63], Usability, Reliability, Documentation and Validity [74].
    Inverse related with Testability [5], Flexibility [5, 33], Portability [5, 33], Reusability [5, 33] and Efficiency.
12. Performance: Direct related with installability.
    inverse related with Reliability and Maintainability.
13. Portability: Reusability [5, 33], Interoperability [63], Interoperability [5, 33, 63] and Efficiency.
    Direct related with Hardware independence [63], Modularity [63], Self Documentation [63], Software system independence [63], Maintanability and Testability.

14. Reliability: Direct related with Computablability [28], Completeness (Correctness Properties) [28], Accuracy [63], Consistency [63], Error tolerance [63], Simplicity [63], Correctness, Generality, Installability, Maintainability [5,33,63], Availability, Usability [5,33,63], Testability [5,33,63], Flexibility [5,33,63], and completeness.
    inverse related with Reusability [5,33,63].

15. Reusability: Direct related with Generality [63], Hardware independence [63], Modularity [63], Self documentation [63], Software system independence [63], Reliability, Usability, Testability, Maintainability, Flexibility and Portability.
    Inverse related with Efficiency, Reliability and Integrity.

16. Testability: Direct related with Instrumentation [63], Modularity [63], Self Documentation [63], Simplicity [63], Corectness, Maintainability, Flexibility [33,63], Portability [33,63] and Reusability [33,63].
    Inverse related with Reusability [5] and Efficiency.

17. Understandability: Direct related with Readability, Complexity, Generality and Modularity.

18. Usability: Direct related with Completeness [28], Effectiveness [28], Communicativeness [63], Operability [63], Correctness, Reliability, Installability, Maintainability, Documentation, availability and Integrity.
    Inverse related with Efficiency, Maintainability [5, 33, 63], Testability [5, 33, 63], Flexibility [5, 33, 63] and Performance.

We summarise all the above interrelation in Table III.1:

	Adaptability	Correctness	Functionality	Consistentability	Modularity	Descriptiveness	Reliability	Understandability	Security	Efficiency	Integrity	Maturity	Suitability	Accuracy	Usability	Communicativeness	Conciseness	Maintainability	Consistency	Testability	Computability	Error-Tolerance	Flexibility	Portability	Completeness	Execution-Efficiency	Reusability	Expendability	Generality	Hardware-Independence	Operability	Self-Documentation	Interoperability	Simplicity	Software-Independence	Storage-Efficiency	Traceability	Training
Adapt.																																						
Corre.																																						
Funct.																																						
Consi.			*R*																																			
Modul.																																						
Descr.																																						
Relia.		*NR*		*R*																																		
Under.																																						
Secur.			*R*																																			
Effic.																																						
Integ.										*R*																												
Matur.							*R*																															
Suita.			*R*						*R*																													
Accur.							*R*																															
Usabi.		*NR*		*R*		*R*	*NR*	*R*		*R*	*NR*																											
Commu.															*R*																							
Conci.																																						
Maint.		*NR*		*R*	*R*	*R*	*NR*			*R*					*NR*		*R*																					
Consi.		*R*					*R*											*R*																				
Testa.		*NR*			*R*		*NR*			*R*					*NR*			*NR*																				
Compu.			*R*				*R*																															
Er-To.							*R*																															
Flexi.		*NR*					*NR*			*R*	*R*								*NR*			*NR*																
Porta.	*R*			*R*	*R*	*R*				*R*								*NR*																				
Compl.		*R*	*R*				*R*								*R*			*R*																				
Ex-Ef.										*R*																												
Reusa.				*R*	*R*	*R*	*R*			*R*	*R*							*NR*		*NR*			*NR*	*NR*														
Expen.																							*R*															
Gener.																							*R*				*R*											
Ha-In.																								*R*			*R*											
Opera.															*R*																							
S-Doc.																		*R*		*R*			*R*	*R*														
Inter.			*R*		*R*				*R*	*R*	*R*		*R*												*NR*								*R*						
Simpl.							*R*											*R*		*R*																		
So-In.																								*R*			*R*											
St-Ef.										*R*																												
Trace.		*R*																																				
Train.															*R*																							

Table III.1: Interrelationships among quality characteristics – R: Related, NR: Non Related, NA: Not Applicable, RR: Reverse Related

APPENDIX IV

QUALITY MODELS

Several quality models have been defined by different people and organizations. "Quality is a multidimensional construct reflected in a quality model, where each parameter in the model defines a quality dimension. Many of the early quality models have followed a hierarchical approach in which a set of factors that affect quality are defined, with little scope for expansion [12]. More recent models have been developed that follow a 'Define your own' approach [29]" [54]. In the following, we summarize briefly some of the most standard and well-known quality models.

IV.1 Hierarchical Models

IV.1.1 McCall's Model (1976)

McCall's model for software quality, see Figure IV.1 combines eleven criteria around product operations, product revisions, and product transitions. The main idea behind McCall's model is to assess the relationships amon external quality factors and product quality criteria.

"McCall's Model is used in the United States for very large projects in the military, space, and public domain. It was developed in 1976-7 by the US Air-force Electronic System Decision (ESD), the Rome Air Development Center (RADC), and General Electric (GE), with the aim of improving the quality of software products" [33].

"One of the major contributions of the McCall model is the relationship created between quality characteristics and metrics, although there has been criticism that not all metrics are objective. One aspect not considered directly by this model was the functionality of the software product" [68].

The layers of quality model in McCall are defined as [17]:

- Factors.
- Criteria.
- Metrics.

IV.1.2 Boehm's Model (1978)

Boehm added some characteristics to McCall's model with emphasis on the maintainability of software product. Also, this model includes considerations involved in the evaluation of a software product with respect to the utility of the program, see Figure IV.2.

"The Boehm model is similar to the McCall model in that it represents a hierarchical structure of characteristics, each of which contributes to total quality. Boehm's notion includes users needs, as McCall's does; however, it also adds the hardware yield characteristics not encountered in the McCall model" [68].

However, Boehm's model contains only a diagram without any suggestion about measuring the quality characteristics.

The layers of quality model in Boehm are defined as [17]:

- High-level characteristics.
- Primitive characteristics.
- Metrics.

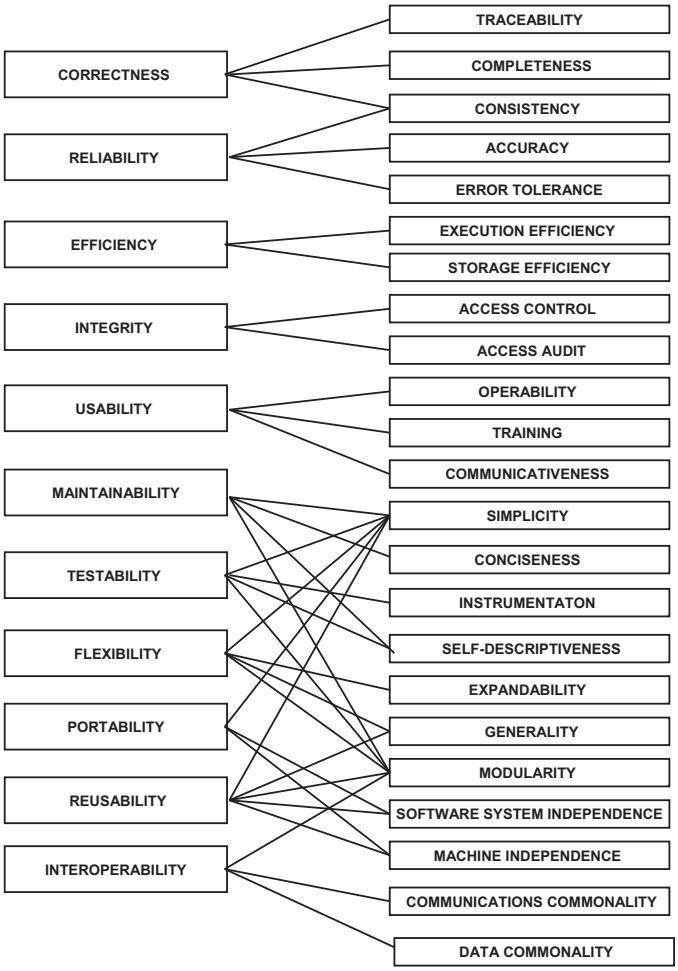

Figure IV.1: McCall's model [71]

IV.1.3 FURPS Model (1987)

The FURPS model proposed by Robert Grady and Hewlett-Packard Co. decomposes characteristics in two different categories of requirements:

- Functional requirements (F): Defined by input and expected output.
- Non-functional requirements (URPS): Usability, reliability, performance, supportability.

Figure IV.3 is an example of the FURPS model. "One disadvantage of the FURPS model is that it fails to take account of the software product's portability" [68].

IV.1.4 ISO/IEC 9126 (1991)

With the need for the software industry to standardize the evaluation of software products using quality models, the *ISO* (International Organization for Standardization) proposed a standard which specifies six areas of importance for software evaluation and, for each area, specifications that attempt to make the six area measurable, see Figure IV.4.

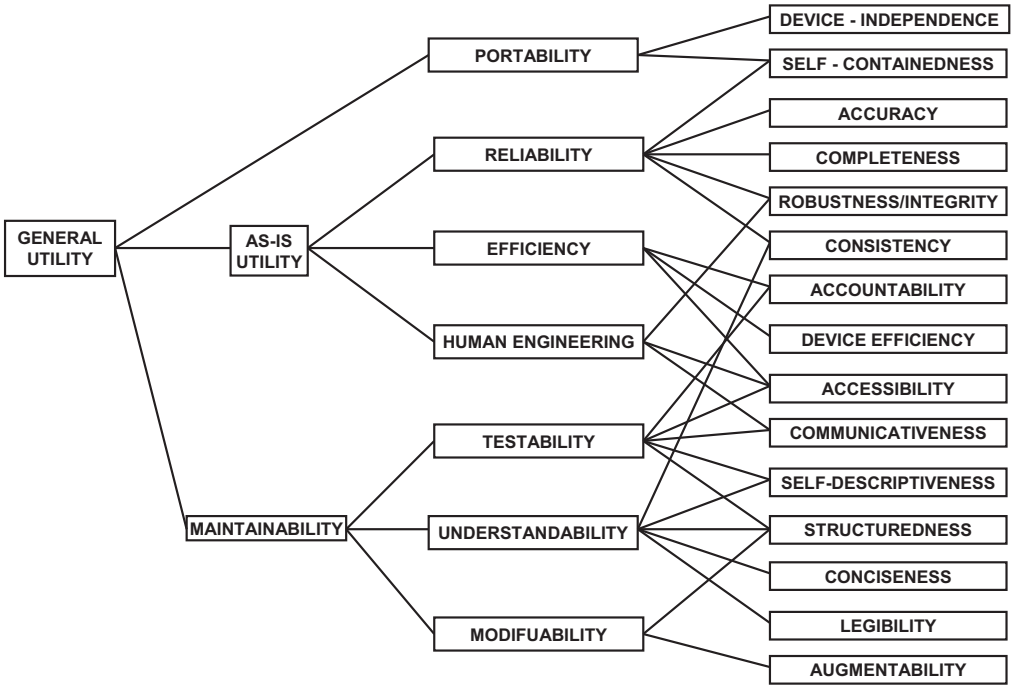

Figure IV.2: Boehm's Model [12]

"One of the advantages of the *ISO* 9126 model is that it identifies the internal characteristics and external quality characteristics of a software product. However, at the same time it has the disadvantage of not showing very clearly how these aspects can be measured" [68].

The layers of quality model in *ISO/IEC* are defined as [17]:

- Characteristics.
- Sub-characteristics.
- Metrics.

IV.1.5 Dromey's Model (1996)

The main idea to create this new model was to obtain a model broad enough to work for different systems, see Figure IV.5. "He [Dromey] recognisees that evaluation differs for each product and you need a more dynamic idea for modelling the process" [28].

Dromey identified five steps to build his model:

- Choose a set of high-level attributes that you need to use for your evaluation.
- Make a list of all the components or modules in the system.
- Identify quality-carrying properties for each component. (That is, qualities of the component that has the most impact on the product properties from the list created in last step).
- Decide on how each property affects the quality attributes.
- Evaluate the model.

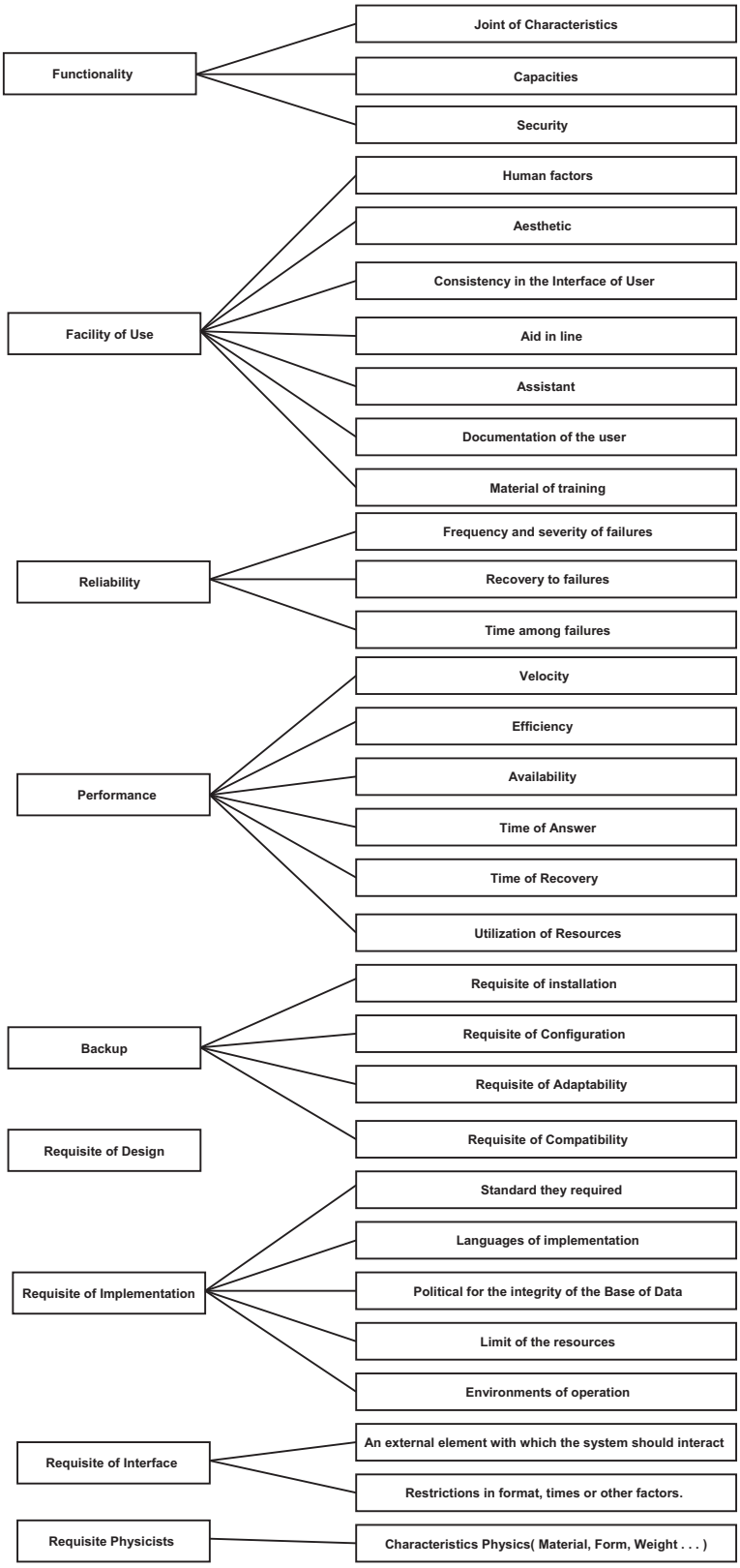

Figure IV.3: FURPS Model

- Identify and resolve weaknesses in with feedback loop.

"Dromeys model seeks to increase understanding of the relationship between the attributes (characteristics) and the sub-attributes (sub-characteristics) of quality. It also attempts to pinpoint the properties of the software product that affect the attributes of quality" [68].

The layers of quality model in Dormey are defined as [17]:

- High-level attributes.
- Subordinate attributes.

Figure IV.6 is an example of a Dromey's model:

- Evaluation of two components (variable and expression).
- Definition of quality-carrying properties for variable and expression.
- Definition of the product properties.
- Obtention of the quality attributes for each product properties from Dromey's model.

IV.2 Non-hierarchical Models

IV.2.1 Bayesian Belief Networks

A BBN[1] is a graphical network whose nodes are probabilistic variables and whose edges are the causal or influential links among the variables. Associated with each node is a set of conditional probability functions that model the uncertain relationship among a node and its parents [64, 65].

Using the BBN have some benefits [64]:

- BBN enable reasoning under uncertainty and combine the advantages of an intuitive visual representation with a sound mathematical basis in Bayesian probability.
- With BBN, it is possible to articulate expert beliefs about the dependencies between different variables and to propagate consistently the impact of evidence on the probabilities of uncertain outcomes, such as future system reliability.
- BBN allow an injection of scientific rigour when the probability distributions associated with individual nodes are simply "expert opinions".
- A BBN will derive all the implications of the beliefs that are input to it; some of these will be facts that can be checked against the project observations, or simply against the experience of the decision makers themselves.
- The ability to represent and manipulate complex models that might never be implemented using conventional methods[2]

IV.2.2 Star Model

The Star model is introduced as follows: "The software quality Star is a conceptual model for presenting different perspectives of software quality. The model is based on the acquirer and supplier as defined in *ISO/IEC* 12207 (1995)" [33].

There are three significant elements in the Star: The procurer (acquirer), the producer (supplier), and the product, see Figure IV.7. The procurer enters in a contract with the producer to create a software product. This contract clearly specifies the quality characteristics of the product. The procurer's perspective of the producer organization is that they use the best project management techniques available and that they engage in first-rate processes to create a quality product. The procurer's perspective of the product is that it must be acceptable by the user community and that it can be serviced and maintained by their professionals.

The model considers that the acquirer be the lead party in any contractual arrangement because it is the acquirer's users and technical support professionals who dictate the success or failure of the software product. Also, it is the acquirer who dictates the profile and maturity of the supplier organization.

"The model accommodates the producer's perspective of software quality and focuses on the maturity of the producer organization as software developers and the development processes that they used to create quality software products" [33].

[1] BBN stands for Bayesian Belief Networks

[2] BBN have a rigorous, mathematical meaning there are software tools that can interpret them and perform the complex calculations needed in their use [64].

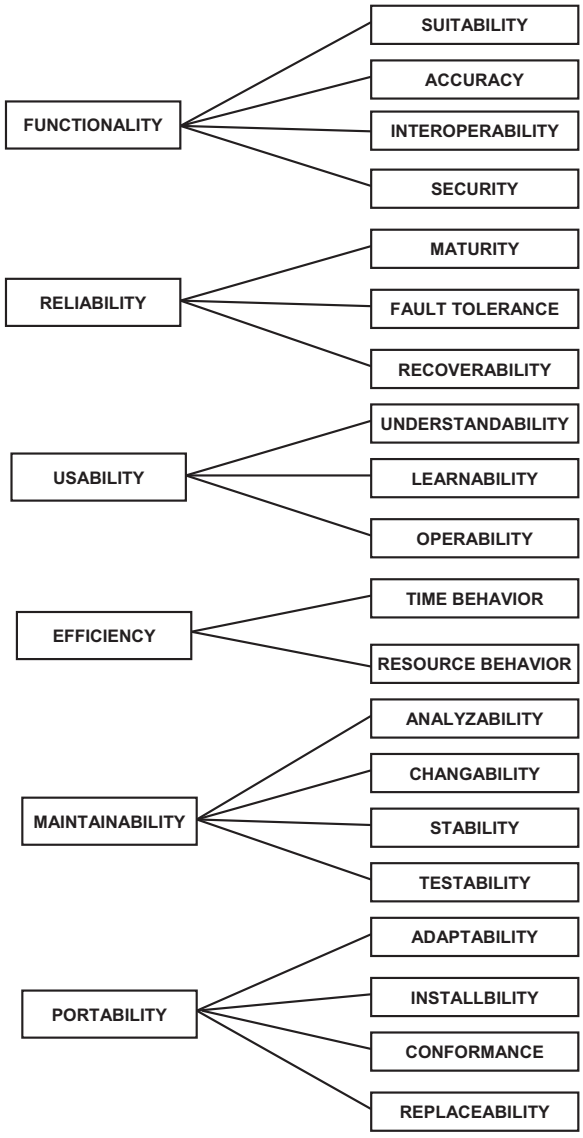

Figure IV.4: Software Quality *ISO/IEC*'s Model

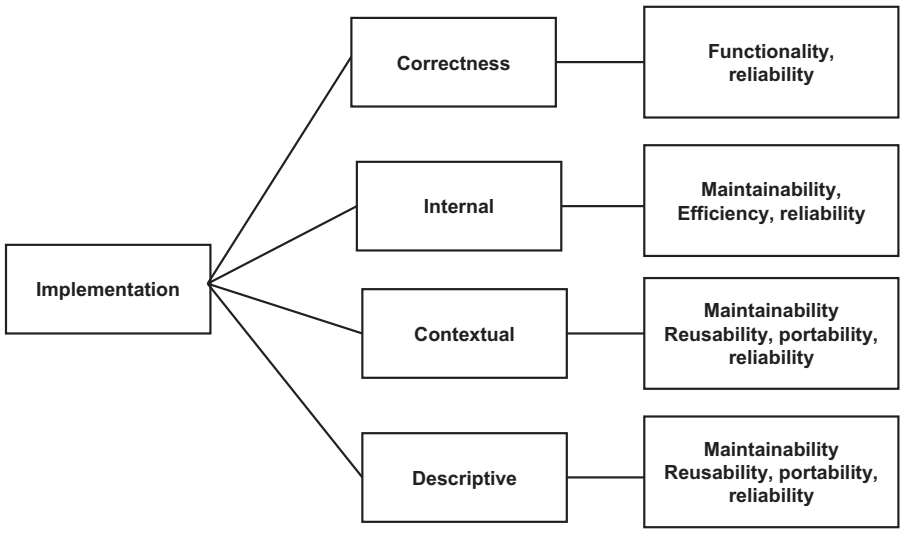

Figure IV.5: Dromey's Model

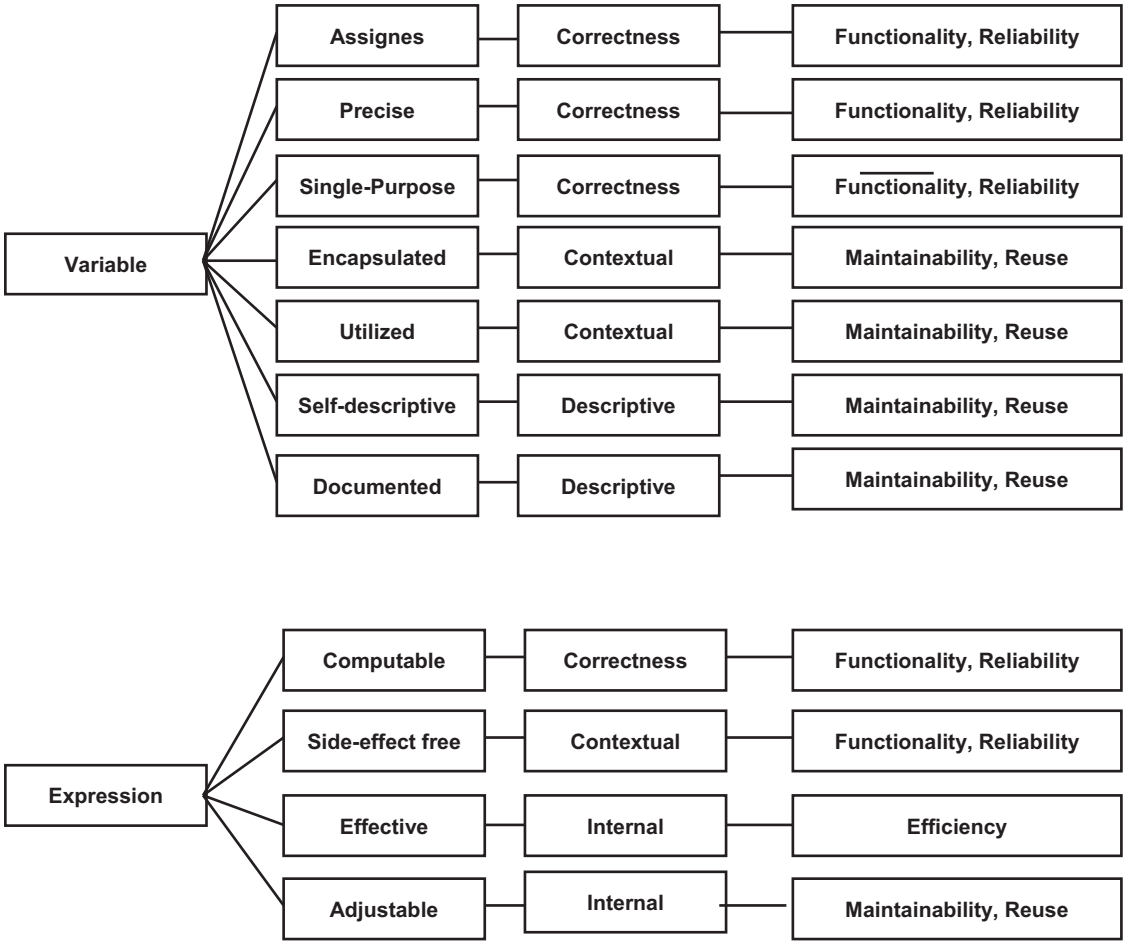

Figure IV.6: Example of Dromey's Model

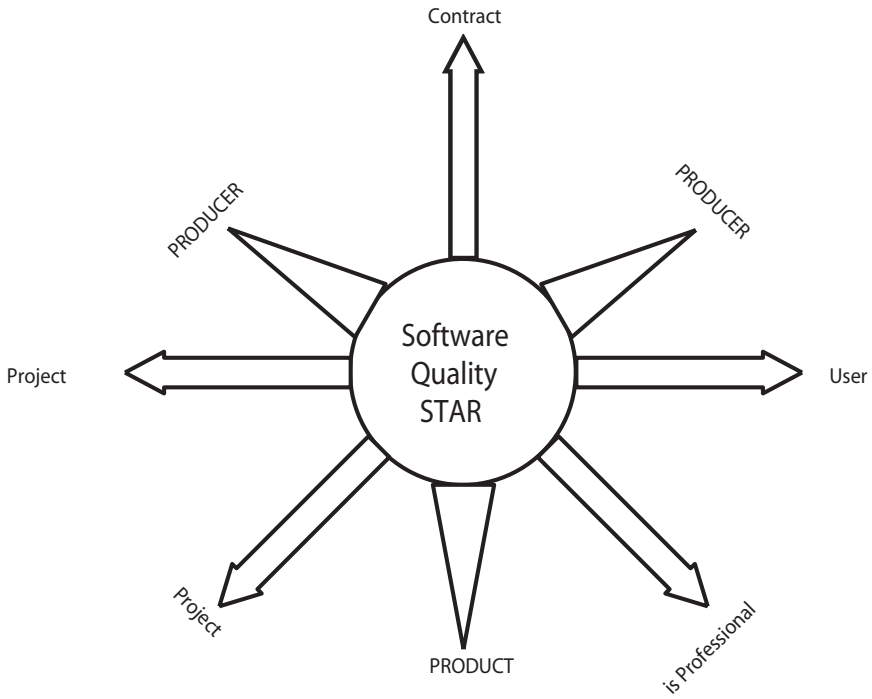

Figure IV.7: Star Model

APPENDIX V

ON QUALITY

The experiences of other people for bring the principle of quality in system and product's cycle could be the most substantial forms of quality evaluation. The following is proposed of few steps, based on experience of different persons about improvement of quality.

V.1 Shewhart

Shewarth [67,80] defined the following step for problem solving and process control in quality.
Plan. Improvement opportunity and outlines the problem or process that will be addressed.
Do. Carrying out the improved process and involves following the plan.
Check. Reviewing and evaluating the result of the changes and determining the effectiveness of the changes to process.
Act. Acting on the analysis and recommended changes.

V.2 Deming

W. Edward Deming [26, 67] was influenced by the work of Shewart [80], and defined the following ideas on quality management:
Constancy of Purpose. Companies face short-term and long-term problem. This requires resources invested in research and development and continuous improvement of existing prodect and services.
Adopt new Philosophy. Lack of purpose and an excessive interest in short-term profits are part of diseases which afflicted companies.
Build Quality in. Performing mass inspections is equivalent to planning for defects and are too late to improve quality, it is necessary to improve the production process to build the quality into the product.
Price and Quality. The price of product or service is meaningless unless there is an objective measure of the quality of the product of service being purchased.
Continuous Improvement. There must be continues improvement in all areas, including understanding customer requirements, design, manufacturing and test methods.
Institute Training. Setting up a training program to educate management and stuff about the company, customer needs, and pride of workmanship in the products or services.
Institute Leadership. Management is about leadership and not supervision.
Eliminate Fear. The presence of fear is a barrier to an open discussion of problem and identification of solutions or change to prevent problem from arising.
Eliminate Barriers. Break down barriers between different department and groups instead of each group optimizing their own area.
Eliminate Slogans. Slogans may potentially alienate staff or encourage cynicism.
Eliminate Numerical Quotas. Quotas act as an impediment to improvement in quality, as quotas are normally based on what maybe achieved by the average worker.
Pride of Work. Remove barriers that rob people of pride of workmanship.
Self Improvement. Encouraging education and self-improvement for everyone in the company.
Take Action. Agreement of management for change and train the staff on the 14 principals.

V.3 Juran

Juran [50, 67] defines quality as "Fitness for use" Juran believe that quality issues are the direct responsibility of management, and the management must insure that quality is planned, controlled and improved. Juran define the Following 10 step programme as roadmap in quality:
Identify Customers. This includes the internal (like the testing group) and external (like the end-users) customers of an organization.
Determine Customer Needs. There may be difference between the real customer needs and the needs as initially expressed by the customer.
Translate. Translating the customer needs into the language of supplier.
Establish Units of Measurement. Defining the measurement units to be used.
Establish Measurement. Setting up a measurement program in the organization and includes internal and external measurement of quality and process performance.
Develop Product. Determines the product features to meet the needs of the customer.
Optimize Product Design. Optimize the design of the product to meet the needs to customer supplier.
Develop Process. Developing process which can produce the products to satisfy the customer's needs.
Optimize Process capability. Optimizing the capability of the process to ensure that products are of a high quality.
Transfer. Transferring the process to normal product development operations.

V.4 Crosby

The main ideas have influenced the Capability Maturity Model (CMM) that created by the Software Engineering Institute [22]. He outlined a 14 step quality improvement program as follow:
Management Commitment. Management commitment and participation is essential to ensure the success of the quality improvement program.
Quality Improvement Team. The representative will ensure that actions for each department are completed.
Quality Measurement. Determine the status of quality in each area of the company and to identify areas where improvements are required.
Cost of Quality Evaluation. Indication of the financial cost of quality to organization.
Quality Awareness. By sharing the cost of poor quality with the staff, it will help to motivate staff on quality.
Corrective Action. Resolving any problems which have been identified, and if it can not be resolved, bring it to the supervisor level.
Zero Defect Program. Communicate the meaning of zero defects to the employees.
Supervisor Training. Manager and supervisor must receive training on the 14 step quality improvement program.
Zero Defect Day. Considering minimum one days in each year to highlight zero defects and its importance to organization.
Goal Setting. Try to getting people to think in term of goal and achieving the goals.
Error Cause Removal. Identify any roadblock or problem which can affect the error free work.
Recognition. Recognizing those employees who make outstanding in quality improvement.
Quality Councils. Using quality professionals for share ideas and action.
Do it Over Again. Continuing is the key of quality improvement.

APPENDIX VI

WEKA: THE WAIKATO ENVIRONMENT FOR KNOWLEDGE ANALYSIS

WEKA is a workbench designed to aid in the application of machine learning technology to real world data sets, in particular, data sets.

In order to do this a range of machine learning techniques are presented to the user in such a way as to hide the idiosyncrasies of input and output formats, as well as allow an exploratory approach in applying the technology. The WEKA machine learning workbench has grown out of the need to be able to apply machine learning to real world data sets.

WEKA Design and Implementation. The WEKA system was designed to bring a range of machine learning techniques or schemes under a common interface so that they may be easily applied to this data in a consistent method. This interface should be flexible enough to encourage the addition of new schemes, and simple enough that users need only concern themselves with the selection of features in the data for analysis and what the output means, rather than how to use a machine learning scheme.

WEKA Design and Implementation. The WEKA system is not a single program, it contain the collection of interdependent programs bound together by a common user interface.

These modules fall into three categories:

Data Set Processing. The processing of data sets involves extracting information about a data set for the user, splitting data sets into test and training sets, filtering out features in the data not required by the user, and translating the data set into a form suitable for a machine learning scheme to work with.

Machine Learning Schemes. Machine learning schemes are implementations of machine learning algorithms and typically take a converted data set and produce some output, normally a rule set.

Output Processing. Output processing modules are concerned with taking the output from a machine learning scheme performing some task with it, such as evaluating a rule set against a test file or displaying the output in a window for the user.