Rev. Fac. Ing. - Univ. Tarapacá, vol. 13 no. 1, 2005, pp. 69-76

# FIVE ONTOLOGICAL LEVELS TO DESCRIBE AND EVALUATE SOFTWARE ARCHITECTURES

**Hernán Astudillo**[1]

*Recibido el 8 de abril de 2004, aceptado el 2 de noviembre de 2004*


## RESUMEN

Los modelos de calidad para la arquitectura del software son taxonomías de atributos de calidad, comúnmente usados para especificar y para evaluar requisitos no funcionales. La mayoría de los modelos de calidad ofrecen un enfoque de dos niveles, que distinguen los atributos externamente observables de los internamente medibles, dando lugar a criterios compuestos de calidad que son específicos a cada stakeholder. Mucho trabajo se ha dedicado a determinar la influencia de los atributos internos sobre los externos, y la mayoría de los modelos usan una jerarquía de dos niveles. Este artículo argüe que este aparente orden dual obscurece el que los requisitos son formulados por diversos stakeholders acerca de sujetos diversos, y que la palabra "arquitectura" significa algo diferente para cada uno de ellos: la organización de un sistema, una descripción de tal organización, y el proceso de elaborar tales descripciones. El esquema propuesto organiza los atributos de arquitectura en cinco niveles ontológicos, que difieren en sus preocupaciones, tipos de usuarios y técnicas de medición disponibles: *computaciones*, *entregables* (binarios/configuraciones), *software* (textos), *especificaciones* (de la arquitectura y/o del diseño) y *proceso de arquitectura*.


Palabras clave: Arquitectura de software, especificación de software, calidad, evaluación, ontología de pruebas, computaciones versus diseño.


## *ABSTRACT*

*Quality models for software architecture are taxonomies of quality attributes, commonly used to specify and evaluate non-functional requirements. Most quality models offer a two-level approach, distinguishing externally observable and internally measurable attributes, yielding stakeholder-specific composite quality criteria. Much effort is devoted to determine which internal attributes influence which external ones, and most models stick to a two-level hierarchy. This paper argues that this apparent dual order obscures the fact that requirements are made by different stakeholder about different subjects, and the word "architecture" means different things to each of them: the organization of a system, a description of such organization, and the process of elaborating such descriptions. The proposed scheme organizes architecture attributes according to five ontological (descriptive) levels, each of them with different concerns, types of users and available measurement techniques: computations, deployables (binaries/configurations), software (texts), specifications (of architecture and/or design), and architecture process. Finally, levels and stakeholders are related to specific architecture views.*


*Keywords: Software architecture, software specification, quality, evaluation, testing ontology, computation versus design.*

## INTRODUCTION

A proposed or existing software system is usually evaluated using long lists of requirements, expressed as attributes, and somehow measuring how well they are satisfied by the system. Beyond merely performing the tasks it was required to do, the system must exhibit several system-wide properties commonly referred to as "the ilities", with names like reliability, maintainability, usability and portability[2]. There are few if any stakeholders of any system who are concerned about all of the above properties. Typically, end users care about observable properties (like functionality, performance and reliability, among others), yet modifiability is paramount in the minds of people who may never use the system at all (e.g. those who would pay for its maintenance or porting).

Software architectures are evaluated with the same criteria formulated to evaluate software products; or rather, evaluating the architecture of a system is taken as

[1] Departamento de Informática, Universidad Técnica Federico Santa María, Avda. España 1680, Valparaíso, Chile, hernan@acm.org

[2] They might as well be called "the nesses", since others are conciseness, completeness and correctness.

evaluating the system it describes. However, this loses sight of the fact that the architect's stakeholders are not necessarily the same as the system's stakeholders. The architect's output is used by developers, QA, system administrators, project managers, and other architects, among others; each audience has specific purposes in mind, and specific quality criteria as well.

An otherwise excellent and lucid treatment of software quality asserts that "the quality attributes that are influenced by the architecture of a system are … functionality, performance, modifiability, and reliability" [1, p.192]. Another widely used text mutters in the same breath "reliability, availability, and maintainability" [13, s.9.4]. In fact, similar statements can be found in most treatments of software architecture quality.

We believe this flat treatment of quality arises from trying to evaluate architecture as if it were one thing, although in practice the term "architecture" is loosely used to describe different things: the organization of a system, the description of such organization, and the process of elaborating descriptions to achieve such organizations[3] . Hence, speaking of "architecture quality" without reference to a specific stakeholder is ambiguous.

Section 2 (Software Quality Models) introduces key notions of quality models, Section 3 (Development of Quality Models) presents some key or currently used quality models, Section 4 (Mistakes, faults and Failures) explains an ontological model used in the testing community, Section 5 (Five Ontological levels) presents a scheme that organizes architecture attributes according to five ontological (descriptive) levels, Section 6 ("Architecture" and Stakeholders) relates these levels to specific stakeholders, Section 7 (Related Ideas) points out some CS developments related to this topic, and Section 8 (Conclusions and Future Work) presents some implications and future work.

## SOFTWARE QUALITY MODELS

Quality models for software architecture are taxonomies of quality attributes, commonly used to specify and evaluate non-functional requirements. Most models (e.g. [5] [11]) offer a two-level approach, distinguishing externally observable and internally measurable attributes, yielding stakeholder-specific composite quality criteria. Significant work has been performed to determine which internal attributes influence which external ones, and most models stick to a two-level hierarchy.

A quality model is a taxonomy of quality attributes and their relationships [1]. A *quality attribute* is a specific characterization or property of a process or product, which can be measured or observed. Quality attributes encompass the traditionally called "ilities". A quality attribute may have several *metrics*, each of which define a measurement or scale (quantitative or qualitative) and a method or technique to observe or measure the attribute. Metrics can be internal or external: internal metrics are applied during construction to the executable system or to its source code (e.g. subsystem performance or code complexity), whereas external metrics are applied to an executing software product (e.g. functionality, reliability and performance).

In most quality treatments, it is mentioned that some quality attributes (like performance and reliability) have quantitative metrics that can be measured by executing the system, and other quality attributes have qualitative values that can be observed by executing the system (e.g. testing for usability) or through non-operational scenarios (e.g. identifying required steps to introduce a new function into the system). There are thus two dimensions of interest: the nature of the measurement (quantitative vs. qualitative) and the time of measurement (executing a system vs. analyzing a description). Clearly, all four possible combinations are in use:

- Executing/Quantitative: performance and stability.
- Executing/Qualitative: usability.
- Analyzing/Quantitative: modifiability
- Analyzing/Qualitative: modifiability and some kinds of performance.

In some quality models, external attributes are correlated with top-level "characteristics" and internal ones with subordinate "sub-characteristics".

Some quality attributes are influenced by the system architecture, and thus can be (partially) evaluated from the architecture descriptions [1]; in particular, functionality, performance, modifiability and reliability. Some system-wide properties, like usability, suffer only minor influence from the architecture.

[3] Yet another use is naming the discipline that concerns itself with these organizations, descriptions and processes; but this meaning is not relevant to this work because requirements made to the *discipline itself* of Software Architecture are well beyond the scope of any development project.

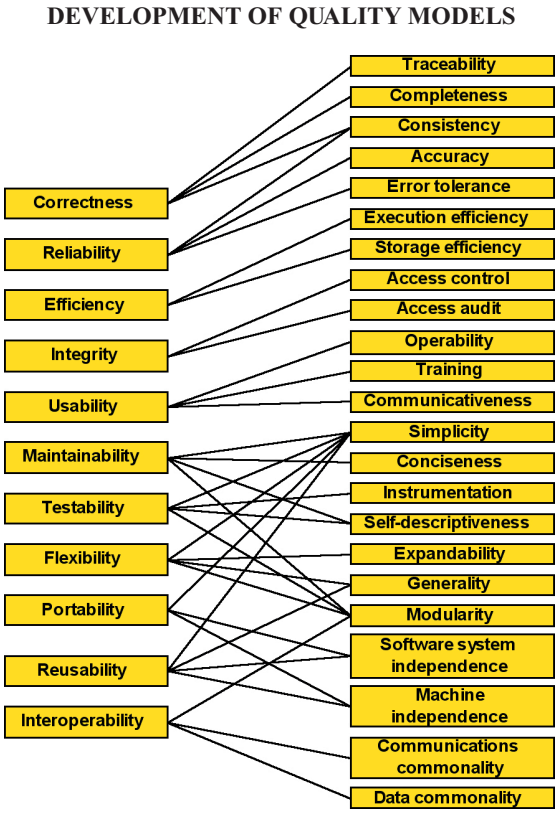


Fig. 1 McCall quality framework.

In the 1970's, McCall and Cavano [12], [5] defined a "quality framework" with criteria for product quality that start off from three points of view (see Fig. 1): [4]

- Operation (using it).
- Revision (changing it).
- Transition (porting it).

About the same time, Boehm [4] proposed a quality model (see Fig. 2) that starts off from refining a system's general usefulness into quality criteria from three points of view, according to the type and circumstance of the users:

- End-user ("as-is" usefulness)
- Potential user, at other place (portability)
- Potential user, at other time (maintainability)

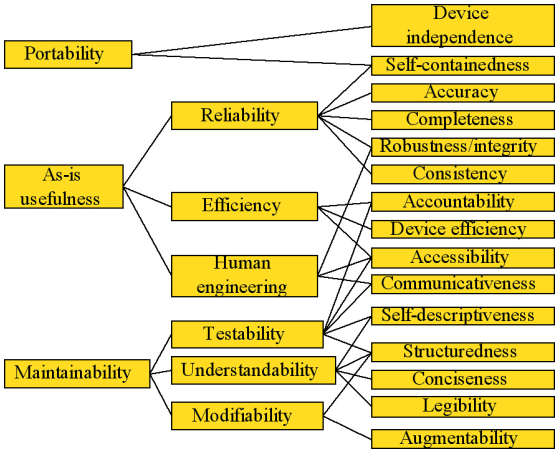


Fig. 2 Boehm´s Quality Model.

Further development of quality models led to a standardization proposal by ISO, in the ISO 9126 [11], which offers a two-level scheme with characteristics and sub-characteristics (see Fig. 3). One key difference of this quality model is that it allows each quality sub-characteristic to influence only one top-level characteristic, unlike most other models.

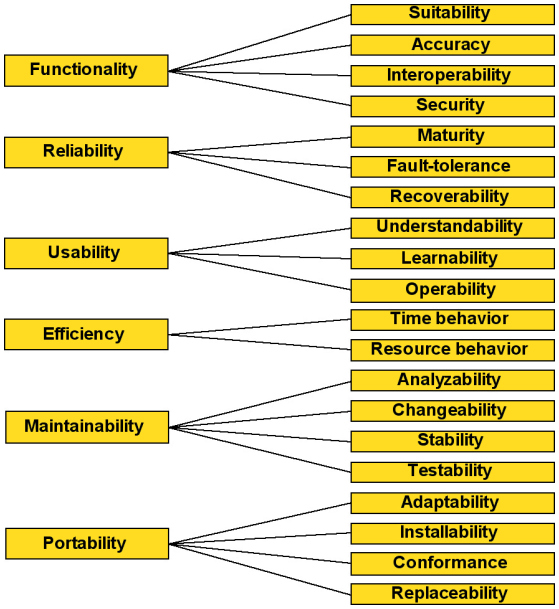


Fig. 3 ISO-9126 Quality Model.

[4] Quality models are herein depicted in a graphical fashion after [13].

A widely used taxonomy for software architecture quality is the one explained by Bass et al. [3], which distinguishes two main kinds of quality attributes according to whether they are observable during execution or not:

Observable during execution

- Performance
- Security and Availability (includes authentication and authorization, MTBF, etc.)
- Functionality (specified tasks)
- Usability (efficient and effective from the user's point of view)

Not observable during execution

- Modifiability (maintainability) (minimal redundancy)
- Portability
- Reusability (total or partial)
- Integrable and easy to test

Other proposed quality models include the one by HP [10], called FURPS (functionality, usability, reliability, performance and supportability), and Dromey's [9] proposal of eight high-level quality attributes, namely the same six from ISO9126 plus reusability and process maturity.

## MISTAKES, FAULTS AND FAILURES

Software testing has a long history already, and this shows especially when considering its careful and mature modeling [13] of its subject matter: mistakes, faults and failures.

- A human *mistake* (error) results in one or more *faults* (defects) in some software product.
- A *fault* may lead to one or more *failures*, if and when conditions for faulty execution are met.
- A *failure* is a departure by the system from its specified behavior (during execution).

Thus, humans incur mistaken actions, which result in faults in software, which cause failures in computations.

The key aspect of this three-level description is that it clearly distinguishes, and yet keeps connected, three domains: *computations*, *texts*, and *action*s. Some implications of the distinction are that different techniques can be used to ferret out these different anomalies; for example, code reviews (involving people) to detect mistakes, fault-seeding (involving software texts) to estimate remaining faults, and stress-test (involving computations) to measure MTBF.

More interestingly to us, the connection between domains leads to techniques that span them; for example, fault-tree analysis [13] allows identifying likely location of faults by analyzing clustering of detected failures.

## FIVE ONTOLOGICAL LEVELS

We would like to generalize testing's three-domain ontology to software architecture (and software design generally), and to this end we must preserve a careful distinction among computations, texts and actions. Clearly, the (architect's) *actions* and the *computations* remain, but unlike the programmer vis-à-vis his programs, the architect is concerned with three kinds of text: elaborates high-level descriptions (*specifications*) of what to build, which are used by designers/developers to build texts (*software)*, which are then translated by automated tools into executable and configuration files (*deployables*). Since systems are collections of programs collaborating to provide the illusion of a single joint behavior, the computational abstractions and externally observable behavior (*computations*) arise from the execution of combinations of deployables to one or more machines.

Thus, we identify five levels, in increasing order of abstraction from the executing system itself:

- *Computations*: the system in execution and the computational constructs it gives rise to, including processes, threads, monitors and so.
- *Deployables*: all artifacts that are installable and/or deplorable for the system operation, including binaries (executable files) and configuration files.
- *Software (texts)*: all textual artifacts that are taken by automatic translators to generate other (probably deployable) artifacts: programs, source code en general, configuration parameters, "makefiles," etc.
- *Architecture specifications*: descriptions of which software must be built, and why; i.e. architecture and/or design specifications.
- *Architecture process*: tasks required to elaborate architecture descriptions.

These five levels are ontologically different, i.e. talk about different kinds of things, and have different organizing quality criteria and evaluation metrics techniques.

One point that may deserve explicit defense are the "deployables;" after all, a single-program developer can (in most cases) assume their existence and correctness. In the architecture process, deployables are what is actually delivered by software organizations to users, and thus what is actually managed by the user's system administrators. Also, since configurations inhabit the gray area of not-quite-automated generation between texts and computations, unlike "executable" files,[5] they must be subject to artifact management even before delivery, just like source code is.

## "ARCHITECTURE" AND STAKEHOLDERS

Requirements come from stakeholders, but not all stakeholders are alike, and not all requirements are alike. Stakeholders make requirements, but not about the same subject. Quality of software architecture is treated as a somewhat global property, but different stakeholders have specific concerns about specific (sub-)products of the architecture process. In fact, there is no single stakeholder who cares for all quality criteria (except of course the hapless architect, who must satisfy them all, and the remote executive common to all stakeholders, if there is one). To wit:

- Although algorithms must be cast in software to be useful, people who design algorithms are almost always concerned with their run-time behavior (time, space, complexity, etc.) rather than their "concreteness" as software artifacts that must be assigned, written, versioned, integrated, deployed, and otherwise managed. Conversely, developers and project managers are usually concerned with these "artifactual" aspects rather than with complexity or integrity considerations; architects are (at most) concerned with picking algorithms or parameterizing their policies; and end-users are concerned with none of this, but only with the resulting behavior and its properties (functionality, reliability, availability, efficiency and so on).

- End users are concerned (usually) only with the "observable" attributes [6], whereas maintainability is a concern of business owners and their operating proxies, the project managers. End-users don't really care for maintainability or portability; they only care about actual, current use of the system. Properties like availability, integrity and confidentiality are closer to their minds, too.

- Scalability, portability and manageability, as evaluable properties, are of concern to the "sysadmins", those who keep the system running and have to adapt it to changing operating conditions. However, usually they *do not* maintain the system, and whoever takes this task will care indeed about maintainability and related criteria.

Clearly, the word "architecture" is loosely used to describe several things: the *organization* of a system, the *description* of such organization, and the process of *elaborating* such descriptions[6]. But the users of the architect's output (a solution and its specification) have specific ideas of what they need and how it is to be evaluated.

A quality model specifically for software architecture (as opposed to software products) could relate quality attributes (e.g. maintainability) to specific subject matters (e.g. software texts) and to specific architecture stakeholders (e.g. project managers). These quality attributes need not be specific, since any of the previously reviewed schemes would provide a good basis for determining what to measure in each case.

To illustrate this approach, we will use a typical set of architecture stakeholders, explored in [2]. A goal of the architect is to *create and document a solution* that pleases a lot of people (his own "local stakeholders", so to speak). Since (with some naiveté) every system-building project can be conceptualized as a modeling effort, with models increasingly detailed and concrete, we start with *analysts* that produce the business model, *architects* who propose major subsystems and technologies, and *developers* who create actual software pieces onto the chosen platform. Each of them has a different take on the solution and its description:

The solution satisfies the requirements of the end user, both functional and non-functional; this should be verifiable by the analyst and by QA. Traceability is a key attribute here[7].

The solution is verifiable by other architects, who can evaluate trade-offs and determine its fitness as solution

[5] An admittedly porous category, since modern Web delivery mechanisms combine several mechanisms.

[6] Yet another use is naming the discipline that concerns itself with these organizations, descriptions and processes; but this meaning is not relevant to this work because requirements made to the *discipline itself* of Software Architecture are beyond the scope of any development project.

[7] Traceability is not addressed explicitly in this description of quality models organizing principles, although in practice is a key quality attribute that influences several high-level criteria.

to the problem. This implies clearly stating the system goals, and offering traceability.

The solution can be built by the developers. This implies partitioning the solution into comprehensible pieces, with clear interfaces and definitions, and explicit mapping of dependencies among pieces.

Yet others are usually ignored as users of the architect's output: *quality assurance* (QA), *project management*, and *administration* of the deployed system[8] :

The product can be tested by QA. This relies on the above mentioned partitioning (to plan unit testing) and traceability (to verify deployed functionality and properties).
The process can be managed by the process manager. This relies on partitioning (to determine work units for teams and individuals) and on dependencies (to schedule work); thus, project managers must be able to determine "intermediate deliverables" that are usable, testable and allow to show working progress.
The product can be administrated by the client's sysadmins. This may imply that the product itself has run-time interfaces for this specific purpose, and that it fits into the environment where it will be deployed (e.g. standard platforms, security schema, integration mechanisms, etc.).

These demands can be organized (see Fig. 4) around the five ontological levels; where possible, we have used the ISO 9126 characteristic name:

The *solution (computations)* must offer the functionality and the observable properties (reliability, usability, efficiency) specified by its requirements (according to users, as estimated by analysts, verified by QA, and predicted by peer architects).
The *solution (executable)* must be installable and operable/administrable (according to the client sysadmins).
The *solution (software)* must be buildable (according to developers and designers), testable (according to QA), maintainable (according to future project managers, as verified by current ones), and measurable by standard tools (according to the project manger).
The *solution description* itself must be analyzable (according to other peer architects).
The *construction process* must be manageable (according to the project manager).

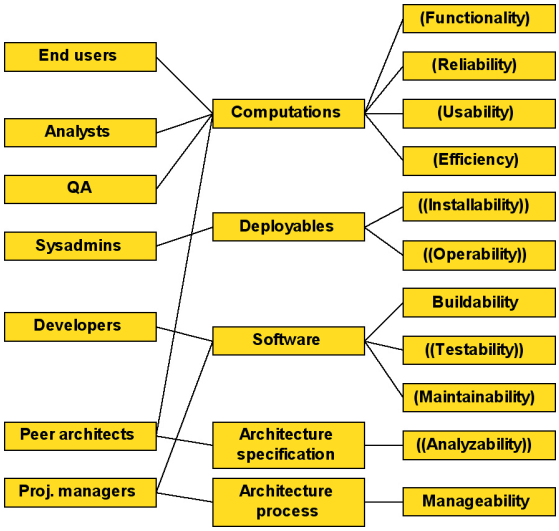


Fig. 4 The Five-Level Ontology and Quality Models.

The resulting combination is shown in Fig. 4, which uses the ISO 9126 as underpinning quality model but can be reformulated as required with other quality models. Attributes shown within single parenthesis (like "(Functionality)") are top-level characteristics in ISO 9126, and those within double parenthesis (like "((Installability))") are second-level ISO 9126 characteristics.

We suggest that clearly identifying the different notions of quality by those concerned with the system architecture will allow stakeholders (and their agents) to recognize quality criteria specific to their role, and gather them accordingly. The absence of such structuring mechanism would, in many cases, lead to trying to settle all quality matters in a process where distinction among requirements must be made by external meta-criteria (e.g. relative importance of their defenders in the organizational hierarchy).

## RELATED IDEAS

The distinction between computations and descriptions of them is also central to a recent proposal by Peter Denning [7] regarding computing matters and practices. Denning argues that we would do well in distinguishing Computing Mechanics (study of computing as matter) and Computing Design (principles of good practice).

[8] Yet other minor roles are omitted here for sake of simplicity.

Computing Mechanics deals with the structure and operation of computations, which are grouped into five categories: computation, communication, coordination, automation, and recollection; some typical topics are Turing machine, grammar, process, naming, caching, and so on. Computing Design gathers principles of design that enable professionals to use their knowledge of Computing Mechanics to build specific systems. The design principles arise from five concerns:

- *Simplicity*: forms of abstraction and structure to hide the apparent complexity of applications.
- *Performance*: throughput, response time, bottlenecks, capacity planning…
- *Reliability*: redundancy, recovery, checkpoint, integrity, system trust…
- *Evolvability*: adaptability to changes in function and scale.
- *Security*: integrity, confidentiality and availability.

Thus, the design principles correspond precisely to top-level attributes in most quality models.

## CONCLUSIONS AND FUTURE WORK

Requirements come from stakeholders, but not all stakeholders are alike, and not all requirements are alike. Stakeholders make requirements, but not about the same subject. It is perhaps unhelpful treating all quality attributes as fundamentally equal, and sweeping away their differences in subject and object as immaterial, or mere "operational differences" to be considered at the time of taking measurements and evaluating quality.

This paper has argued that current quality models based on two-level schemes of quality attributes are sidelining two core issues regarding usefulness of such models: requirements are made by different stakeholder about different subjects, and the word "architecture" means different things to each of them (organization of a system, the description of such organization, and the process of elaborating such descriptions). To address these issues, it proposed a five-level description for software architecture outputs, each level with different concerns, types of users and available measurement techniques: *computations*, *deployables* (binaries and configurations), *software* (texts), *specifications* (of architecture and/or design), and the *architecture process* itself.

Work currently in course aims to extend to the proposed 5-level model some implications of the mistake-fault-failure model from testing, such as the predictive/ analytical models that have arisen from combining levels. A central concern is trying to determine the equivalents for "bug" in architecture, a question that itself requires disambiguation since all five ontological levels necessarily will have different notions of bug.

Finally, we suggest that clearly identifying the different "things" desired and measured by those concerned with the system architecture will allows stakeholders (and their agents) to recognize quality criteria specific to their role, and gather them accordingly, instead of trying to settle all quality matters as a process where distinction among requirements is to be made by external meta-criteria (e.g. relative importance of their defenders in the organizational hierarchy).

## REFERENCES

[1] S. Albin. "The Art of Software Architecture: Design Methods and Techniques". Wiley (2003).

[2] H. Astudillo and S. Hammer. "Understanding Architecture: What we do and why we do it" OOPSLA'98 Workshop on Architecture as Method. Vancouver, BC, Canada, Oct. 1998.

[3] L. Bass, P. Clemens and R. Katzman. "Software Architecture in Practice". Addison-Wesley, 1998.

[4] B.W. Boehm, J.R. Kaspar, M. Lipow, G. MacCleod. Characteristics of Software Quality. Amsterdam: North Holland.

[5] J.P. Cavano and J.A. McCall. "A Framework for the Measurement of Software Quality." Procs. ACM Software Quality Assurance Workshop, pp.133-139, Nov. 1978.

[6] P. Clements, R. Katzman and M. Klein. Evaluating Software Architectures. Addison-Wesley, 2002.

[7] P. Denning. "Great Principles of Computing." Communications of the ACM, 46(11):15-20, Nov. 2003.

[8] L. Dobrica and E. Niemelä. "A survey on Software Architecture Analysis Methods." IEEE Transactions on Software Engineering 28(7): 638-653, July 2002.

[9] R. Geoff Dromey. "Cornering the Chimera", IEEE Software, 13(1):33-42, January 1996.

[10] R.B. Grady and D.L. Caswell. Software Metrics: Establishing a Company-Wide Program. Prentice-Hall, 1987.

[11] ISO/IEC 9126. Information Technology - Software Product Evaluation: Quality Characteristics and Guidelines for Their Use, 1991.

[12] J. McCall, P. Richards and G. Walters. Factors in Software Quality (3 vols.), NTIS AD-AO49-014, 015, 055, Nov. 1977.

[13] S.L. Pfleeger. Software Engineering: Theory and Practice (2nd Ed). Prentice Hall, 2001.