# Software Quality Modelling in managerial decision support system

Doctoral Qualifying Examination

**Ivica Ćubić**

Ericsson Nikola Tesla d.d.
Research & Innovations

European Commission
Future and Emerging Technologies
ivica.cubic@ec.europa.eu

Postgraduate study in
Electrical Engineering and Information Technology

Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture
University of Split
Croatia

## Contents

# 1 Introduction

Software is ubiquitous in all areas of living and industry. It is driving mission-critical operations, and its quality is certainly an issue of paramount importance. Majority of software products and systems is commercial software viewed through its major parameters of quality, time-to-market and associated costs. Software quality is considered in different disciplines of software engineering, such as software development and requirements engineering, and different phases of software development. It is, on the other hand, rarely considered as a cross cutting issue across all disciplines of software engineering, including project and business perspectives. Moreover, scientific considerations have seldom been synchronized with industrial practices and issues, while industrial approach generally has not followed scientific breakthroughs [1].

Since software development is not conducted in a controlled and predictable way, and to some extent depends on crafting, a sound managerial decision support system is needed. An advanced decision support system with comprehensive software quality model will help in better understanding and description of decision problems, better adhering to requirements, timely reactions, decision communications and development of better software tools and systems. The need for decision support exists during the complete software life cycle [2]:

- Requirements, concerning the time and budget constraints functional and non-functional requirements should be chosen [3]
- Project management, original project plan in terms of budget, time and quality should be monitored including reaction on possible deviations
- Maintenance, which modules are error prone to concentrate quality assurance efforts.

This paper overviews the software quality issues and focuses on software quality modelling with an emphasis on industrial issues and needs. Like other engineering and scientific disciplines, software engineering also uses models to understand and control software quality issue [4]. The existing software quality modelling approaches will be classified and overviewed.

The overview of software quality modelling will start with internationally accepted and standardized definitions and taxonomies with a view to establish a common software quality communication dictionary. Integral approach to software quality modelling calls for inclusion of all relevant entities. Besides the software products themselves, other important components are processes and resources. Usual, product-based approach will be accompanied with other general product quality approaches and explored in order to provide reliable software quality assessment and prediction thereof. This paper will start up the work of application of general product notions of quality based on product, user, manufacturing and value based approaches [5] to the software respecting all software particularities. The main software characteristics under observation will be reliability, maintainability and usability because they comprise the well studied and important phenomena of software defects and removal, high costs related to software maintenance and user acceptance of software products and services.

The software quality model will be put in broader managerial support system framework, comprising other important aspects i.e. time and resources, using Multi-Criteria Decision Making (MCDM) methodology. The whole framework should provide quantitative support for managerial decision-making in software development emphasizing software quality issues. The advanced decision support system should be based on sound methodology, best practices, and empirical validation. Methodologies applicable in decision support are modelling, measurement, simulation and knowledge management. Moreover, an optimal decision support system could be built by a hybrid approach integrating different methodological contributions. The general managerial decision support framework could be further simplified and customized by targeting the telecommunication control software domain and data availability in order to assess its applicability and usefulness in practical industrial software development and further research.

The second chapter will bring the software background issues related to software quality as well as current, quality related trends in the telecommunication industry. Software modelling as an approach to understand, integrate, control and predict software quality will be overviewed in the third chapter. The fourth chapter will explore contemporary, soft computing approach and integration of software quality models into a broader support system for managerial decision during software development. At the end the conclusions will set further research issues and directions.

# 2 Software engineering and quality

In order to broadly and integrally address software quality issues, major related software engineering disciplines will be overviewed. The overview is started with a general overview on product quality [5] and software quality particularly [6] [7]. Then we will proceed with software project failure rate issue raised by CHAOS reports [8] and other relevant software engineering procedures, such as software product life cycle and software processes. The chapter finally brings some observations of software quality in telecommunication industry.

Software quality itself is a concept embracing different qualities that will be described in this paper. Those particular qualities are perceivable entities that characterize software artefacts such as software modules, products or whole software based systems and they are related to other qualities reflecting hierarchical order or quality interdependencies. To deal with such abstract and elusive concept, different techniques can be used in conceptualization of software quality i.e. representation of software quality concepts similarly to software functional requirements [4, 6, 7]. The examples of such techniques are quality modelling [4], meta-modelling [9] or quality ontological engineering [10]. Models formally describe problem domain to facilitate understanding and communication among stakeholders and to capture the knowledge about software quality.

## 2.1 Software quality

Software quality could be defined in many ways. The good starting point is a general observation on product quality in different domains [5]. According to Garvin's influential article quality can be defined in five different ways:

- Transcendent approach of philosophy;
- Product-based approach of economics;
- User-based approach of economics, marketing, and operations management;
- Manufacturing-based;
- Value-based approaches of operations management.

Traditionally, consideration of software quality puts a focus on code quality and defects then on the software development processes (corresponding to the general product manufacturing-based view), but value-based (economics/costs) and user-based (the end customer or user satisfaction) becomes also extremely important.

The emphasis on software defects is clearly justified by some estimations that 50% of software development costs attribute to defect-detection and removal [11]. Applying the value-based approach, quality requirements and characteristics such as reliability, could been also optimized, investing (effort i.e. person months and time) only in necessary and agreed quality.

Besides value-based view of software quality, the other important view is user-based view. The user-based view is grounded in satisfying the user's needs view and it is inherently included in functional suitability, reliability, performance efficiency and usability [7]. User-based view becomes extremely important with new business models such as cloud and software-as-a-service (SaaS). The user-based view for general products is well known [5], but for software products is not well studied, especially not by the scientific community.

On the other side there is open source software. The open source software addresses some community or general public needs for software, and maintenance or generally responsiveness to defects fixing depends on voluntary contributions and is hardly predictable. That is completely unacceptable for mission or business critical software applications. Interesting situation is, where due to time and budget constraints, we have a mixture of commercial and open source software and how software quality of such products can be addressed. Besides quality assessment and assurance, with such systems that comprise both commercial and open source software, legal and business issues should also be carefully considered. Maintenance and generally support of open source software could also be commercially offered service, but that will lead to a distributed and partly outsourced quality assurance.

Traditional approach to software quality has treated software quality as intrinsic characteristic of a software product. That assumption is similar to notion of software laws i.e. various software systems and processes abide by the physics-like laws. Here a "physics-like law" means that an invariant pattern can be applied to a variety of software behaviours with high precision. Some examples of software laws are [12-15]:

- The observation that 20% code of a software system are executed at 80% times,
- The Pareto principle, that a small number of modules contain a majority of defects,
- The early fault data can be used to predict later fault and failure data,
- The larger modules have a lower fault density than smaller ones,
- Software metrics such as size (lines-of-codes) and code complexity metrics are good predictors of fault and fault-prone software modules.

Contemporary view on software quality challenges intrinsic software quality as well as software laws. As the number of observed faults/defects and failures associated with a software module depends also on the amount of testing and time under operations, there is no clear evidence neither on causality between software metrics and quality characteristics nor that software metrics are good predictors of faults/failures [14]. The software quality is seen as a combined influence of various properties of the product and user based view on product economics [16].

## 2.2 Software project failure rate

Due to software development intrinsic problems [17] and despite the effort and research on methodology and processes, software project failure rates have not decreased significantly. Neither has there been any proven, significant and systemic improvement in software quality [18]. A software development project could be assessed on basis of the cost, time, or quality of an outcome as in CHAOS reports [8]. On the high level assessment there are cancelled and non-cancelled i.e. finished software projects. The software projects cancelation rate are mostly just below 20% with decreasing trend over time.

Product quality, for projects not cancelled, is one out of five success criteria measured by [18] and four others are:

- User satisfaction,
- Staff productivity,
- Time-to-market,
- Budget.

In that measurement, product quality is - along with user satisfaction and staff productivity - perceived either poor or fair (4 point, forced, Likert scale: poor/fair/good/excellent) in approximately one third of not-cancelled projects.

The most critical problem in finished software projects is estimating and managing the schedule (time-to-market). The second critical performance problem is budget. For cancelled projects critical quality problem is on the bottom of the reason for cancellation list with 1.35% while on top of the cancellation list are not sufficient involvement of senior management, too many requirements and scope changes, and overspending. Cancellation reasons should be taken with caution because they are calculated on sample of 18 cancelled projects in 2007 [18]. Moreover, the software quality in CHAOS assessments is perceived narrowly, and with broader, integral, definition of software quality, user satisfaction could also be included under software quality issue.

Software quality is not a top issue either for cancelled or non-cancelled software projects. That is a very good argument for necessity of inclusion of cost and time parameters for sound managerial decision support system.

Moreover with increasing of software presence in the mission critical applications and dependability of those applications on their software components, software quality is an inevitable issue. For software project managerial decision support applications and accompanied models it is important to embrace all relevant views i.e. time, cost and quality of outcomes, as well as to harmonize and integrate all involved disciplines such as science, engineering, management and finance [19].

## 2.3 Software product life cycle

An integral software quality model should address a complete software product lifecycle and environment. Bøegh emphasized the importance of the system perspective for eliciting software quality requirements [20]. He outlined a simple, hierarchical model of system needed to describe software quality from the system perspective. The model comprises:

- Communicating computer systems, each of them comprises hardware, software (the operating system and applications) and data
- Mechanical parts (to include also embedded systems) such as mechanics, hydraulics
- Human processes (not everything is automated and at least to cover business and policy decisions).

Since the complexity of human-made systems comprising software elements has significantly increased a common framework to improve communication and coherent cooperation between stakeholders is absolutely needed. Standards ISO/IEC/IEEE 15288 System Life Cycle Processes [19], and ISO/IEC 12207 Software Life Cycle Processes [21] cover complete life-cycle processes of general man-made systems and software, respectively. IEEE cooperated on and adopted both standards. ISO/IEC/IEEE-15288 describes the life cycle of human-made systems from an idea through to the retirement of a system by defining a set of processes and associated terminology. For system comprising software, the ISO/IEC/IEEE 15288 is supplemented with the ISO/IEC 12207 standard. The ISO/IEC 12207 standard provides a comprehensive set of life cycle processes, activities and tasks for a software that is part of a larger system, and for stand-alone software products and services [21]. The standard ISO/IEC 12207 is drawn as a framework for understanding and cooperation of stakeholders in the international software market.

The two standards are aligned and designed to be used together for software intensive systems. The ISO/IEC/IEEE 15288 describes the processes at the system level, while the ISO/IEC 12207 specializes the same processes to software, and amends software specific processes.

The ISO/IEC/IEEE 15288 differentiates (Figure 1):

- Agreement processes (Acquisition and Supply) as interfaces toward other organizations
- Organizational project-enabling processes such as Infrastructure, Human resources and Quality management

- Project processes, primary such as Project planning and supporting such as Risk management and Measurement
- Technical or engineering processes such as Implementation, Operation, Maintenance and Disposal.

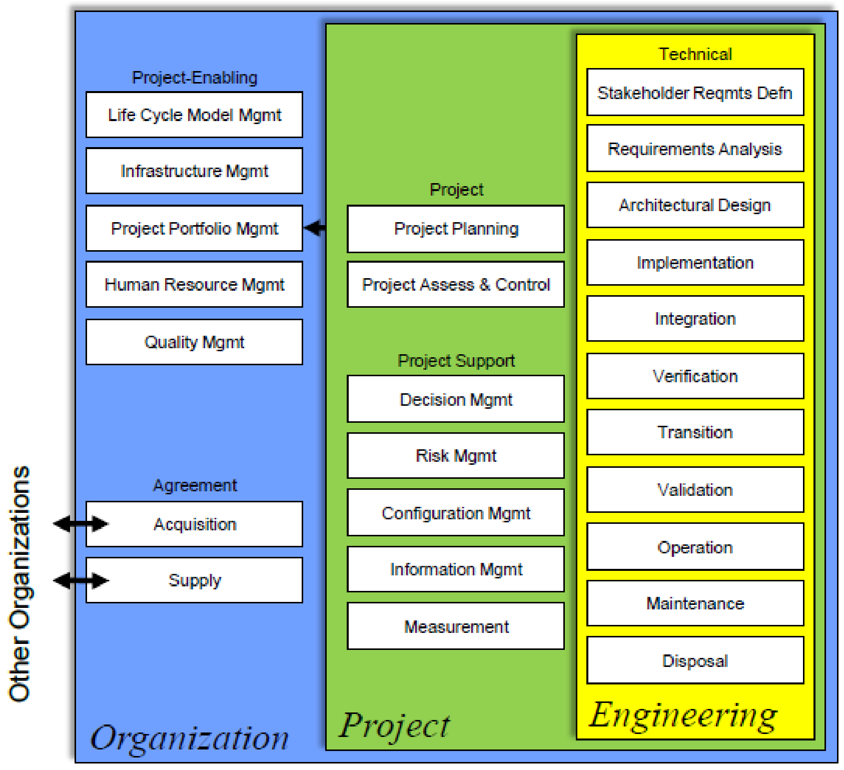

**Figure 1 The ISO/IEC/IEEE 15288 overview**

The ISO/IEC 12207 further describes Implementation as Software implementation comprising also Software quality testing (Figure 2). It adds Software support processes such as Software quality assurance, Software verification (a software meets its specifications) and Software validation (a software fulfils its intended purpose). The third group of the ISO/IEC 12207 processes tackles Software reuse.

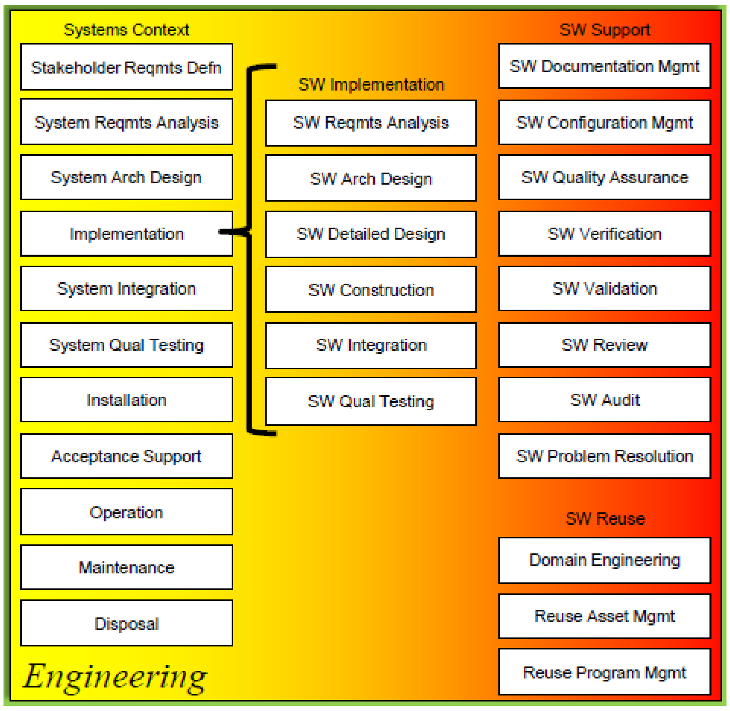

**Figure 2 The ISO/IEC 12207 complements the ISO/IEC/IEEE 15288 concerning software life cycle**

Processes in these International Standards form a comprehensive set from which an interested party can construct system life cycle models appropriate to its applications, products and services. Depending on the purpose, an appropriate subset of processes can be selected and applied to fulfil that purpose.

There are six core processes (primary lifecycle processes) in creating a software product:

- Acquisition,
- Supply,
- Implementation (Development),
- Operation,
- Maintenance,
- Disposal.

For software quality purpose, an abstract or general software product lifecycle, abstracting the differences due to different approaches (two major approaches are classical or waterfall and agile or iterative) will be useful.

## 2.4 Software processes

There are different approaches how companies define, perform and institutionalize their quality management processes. Besides the software product quality, there is also an approach that software quality can be reliably and trustworthily assessed by evaluating the processes applied in software production. Standards ISO 9001 [22] and Capability Maturity Model Integration-CMMI [23] focus on the quality of the software processes [24].

The manufacturing view [5], originally described as conformance to the specification, could be for the software quality purposes understood as process view. Three major approaches to software development are classical, sequential and plan-driven, waterfall model; iterative and prototype oriented Rapid Application Development (RAD) model [25]; and agile methodologies including Scrum as combination of incremental and iterative model [26]. Sutherland asserted that combining of CMMI with Scrum improves software quality [27].

The waterfall model organizes the software development lifecycle into five linear stages:

- Requirements analysis and definition,
- System and software design,
- Implementation and unit,
- Integration and system testing,
- Operation and maintenance.

It is the oldest and the most mature software development model. The waterfall model shows its advantages with large and complex systems, but has also a number of disadvantages such as inflexibility when facing changing requirements and high complexity irrespective of the project size [28].

RAD model is trying to speed up the development with lowering costs and improving quality. The focus is on prototyping and user involvement.

The main advantages are easiness of implementation and short time-to-market, while main disadvantages are a real threat of bad design due to the speed development and necessity for strong control and project management [25].

Agile development moves focus from detailed planning and plan-based control to change management and collaboration between developers and customers [29]. Agile software development comprises different practices and methods. The most know and adopted Agile methods are:

- Extreme Programming (XP) with focus on implementation,
- Scrum with focus on agile project management.

In Scrum, a self-managing team develops software in increments. Those increments are called sprints and last from one to four weeks. Wanted system features are registered in a backlog. The product owner decides which features from the backlog should be developed in the following sprint, depending on estimated feature effort. A sprint starts with planning and ends with a review. Team members coordinate themselves in daily stand-up meetings [26]. The important characteristic is that a previous sprint should finish with a simpler, working version of the system that will be enhanced in the following sprint. Such a sprint or iteration can be also treated as a miniature waterfall life cycle [28].

As Agile development moves from academic and education institutions towards mainstream software development community and professional software development organizations [30], some studies compare software quality regarding the applied process. Classical approach i.e. waterfall model has quality assurance activities in parallel with development. The contemporary, agile approach integrates software quality assurance with development by test driven development combined with Scrum methodologies.

Huo compared the software quality assurance activities in waterfall model (Software Quality Assurance-SQA and Verification and Validation-V&V) and Agile approach (pair programming, refactoring, continuous integration and acceptance testing) [28]. The conclusion was that Agile approach may lead to better software quality and shorter time to market due to tight integration of quality assurance practice with the development phase, higher frequency of occurring quality assurance practices and their application from the earlier stages.

Li conducted a longitudinal case study on software quality with a transition from a plan driven process (17 months) to Scrum (20 months) [26]. The conclusion was that Scrum may not lead to a lower defect density than a plan-driven process, but defects are detected and fixed much earlier due to the short sprints (four and two weeks) and the defect fixing efficiency could be improved due to daily Scrum meetings and knowledge sharing. All that comes at the cost of increased stress of Scrum developers to deliver on time and within budget, which could also make developers reluctant to take care of maintainability.

CHAOS Manifesto [8] is advocating the agile process, stating that the agile process has built-in quality and is test-driven.

## 2.5 Software quality in telecommunications

In telecommunications industry with current major trends of hardware virtualization, software defined networking (SDN), cloud computing and software-as-a-service (SaaS), software products and systems take a significant share of total investment and became the major component of telecommunication mission critical operations.

Telecommunication systems shall have the availability of *five nines* 99.999% which means 5.26 minutes down-time per year with planned upgrades and maintenance. Telecommunication system providers strategically perform quality assurance activities to ensure the required level of quality [24].

Cloud business model and SaaS paradigm induce shifting of focus toward end users. In order to enhance customer experience service and support, the ability to rapidly resolve any performance or technical issue is very important. Common customer-driven key quality indicators are:

- Customer experience index for voice
- Customer experience index for data
- Data connection failure rate

Those customer-centric indicators have to be further translated into specific network and IT indicators. That will enable service providers to drill down from customers experience issues to the infrastructure level, including software support, for troubleshooting. Each system service session should be mapped to the network and software resources that contribute to delivering the service. To identify the root cause of user experience quality problems and bottlenecks, the essential tool is monitoring function. Such monitoring function depends on detailed event reporting from the network. The relation to software defects or general software quality is mainly indirect, i.e. after detection of cumbersome network functionality, responsible software drivers should be identified and issue fixed.

The trend with the telecommunications services is direct interaction of customers with the network: subscription to a service, changes of service levels and subscription cancellations should be done directly by end users in a few minutes. The conformance to a Service Level Agreement becomes also essential and should be described by technical and quality characteristics translatable to customer quality indicators.

Software becomes an integral part of various service systems, therefore the software quality is somehow reflected through the quality of service. In a holistic approach to software quality, the user experience (UX) becomes the most important measure of software quality, and user experience is closely coupled with quality of service (QoS).

## 3 Software quality modelling

There is a number of models treating software quality but with different approaches. Thus the software modelling classification has become very important. Fenton and Neil simply observes software models and their applications as a comprehensive software metrics [1].

The main applications and use of models are predictions of resource requirements and quality predictions [31]. The big amount of quality prediction models is defect prediction models. Defect prediction models comprise also classification models. The classification models predict a software module as fault-prone (*fp*) or not fault-prone (*nfp*). In order to timely identify fault-prone modules, before the testing so the tests could be focussed and optimized, most of the classification models are based on software metrics. Common approaches for exploiting software metrics data is data mining and machine learning, while common classification techniques are decision rules and decision trees (C4.5 and Random Forest) based models [32, 33].

A first, broad qualification of software models was proposed by [34]. The classification was based on level of specialization of so called quality-evaluation models:

- Generalized quality-evaluation models,
- Product specific models.

The generalized models are further divided on overall, segmented and dynamic. A segmented model estimates quality for different industrial segments, and an example of segmentation is reliability level (Table 1).

**Table 1 A segmented model for reliability level estimation (Tian 2004)**

| Product type | Failure rate (per hour) | Reliability level |
|---|---|---|
| Safety-critical software | $< 10^{-7}$ | Ultra-high |
| Commercial software | $10^{-3}$ to $10^{-7}$ | Moderate to high |
| Auxiliary software | $> 10^{-3}$ | Low |

Telecommunications software, according to Tian [34], belongs to commercial software segment, while Wood [35] considers it critical business application. Telecommunications software is not so critical as military or space software applications, but their importance as basic services for the non-telecom businesses and the complete society, makes telecommunications software more critical than the rest of commercial software. Five nines or 99.999% of required availability for telecommunications software means 5.26 minutes of downtime per year or translated in failure rate (per hour) that allowed just a fraction of failure or few failures depending on their severity and recovering pace [24]. If you have a serious failure that will stop a telecommunications software for a half day (12 hours), the failure rate per hour will be $8 \cdot 10^{-7}$ without any other failure.

The product specific models provide more precise evaluations due to usage of product specific data. Tian further divides the product specific models to [34]:

- Semicostumized models use extrapolation of product history to predict quality for the current project, and as such are suitable for software products with more releases. An example of the group is the orthogonal defect classification (ODC) model [36, 37].
- Observation-based models estimate quality according to current project observations e.g. observed defects and associated time intervals are fitted to software reliability growth models to evaluate product reliability like in the Goel-Okumoto model [38].
- Measurement-driven predictive models establish predictive relations between quality and historical measurements either using statistical analysing techniques or learning algorithms. An example is tree-based model to analyse the relationship between defects fixes and various design and code measures [34] in order to identify error prone modules and focus inspection effort on a few selected modules.

As telecommunications software industry mainly deals with software system releases, usually of evolutionary nature, the orthogonal defect classification (ODC) model could be successfully applied and even used to sufficiently quantify the key cause-effect relationships for further software development process improvement.

Another software models' classification scheme is based on different purposes of numerous software quality models, namely definition, assessment, and prediction of quality [4]. Definition, assessment and prediction are not independent of each other and those mutual dependences between different model classes and real software quality models are depicted in Figure 3.

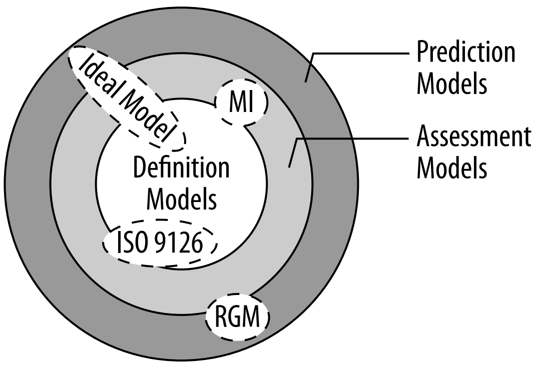

Figure 3 Definition, Assessment, prediction (DAP) Classification for quality models [4]

An ideal model features prediction model as the most advanced form of quality models, but it can also be used for definition and assessment of quality. The definition-asses-prediction (DAP) classification will be used throughout this paper to overview existing software quality models.

## 3.1 Software definition modelling

A common language or taxonomy is needed to establish common dialogue and an understanding on software quality issues. The taxonomies are usually brought by the software quality definition models and standards. Quality definition models comprise the set quality characteristics and the relation between them as basis for specifying quality requirements and evaluating product quality. Those quality characteristics should reflect the overall quality of the software product [39, 40].

The common and accepted software definition models are:

1. McCall (1977)
2. Boehm (1978)

3. FURPS (1992) Grady and Hewlett Packard
4. ISO/IEC 9126 (1991)
5. Dromey (1995)
6. ISO/IEC 25000 – SQuaRE (2005)

McCall in his handbook on software quality emphasized the contribution of the software quality metrics to a more disciplined, engineering approach to a software quality assurance [41]. He applied the so called factor-criteria-metrics (FCM) approach in order to bridge the gap between users and developers. Quality factors (characteristics) describe the external or user view and have a set of quality criteria describing the internal or developer view. Each quality criteria has one or more metrics.

Boehm brought a hierarchal structure similar to McCall, comprising high, intermediate and low level of software quality characteristics [42, 43]. He additionally emphasizes the maintainability characteristic [40] but does not suggest any approach to measure their quality characteristics [39].

FURPS model [44] differentiates functional and non-functional requirements. Functional requirements related characteristic (F - Functionality) are defined by input and expected output. Non-functional requirements related quality characteristics are Usability, Reliability, Performance, and Supportability (URPS).

ISO/IEC 9126 [45] international standard suite represented a high-level framework for characterizing software product quality. The standard was considerably founded on the Boehm model [46]. It decomposed software quality into six characteristics, and these characteristics are further decomposed into subcharacteristics. Quality subcharacteristics are results of internal attributes and are manifested externally during the software product usage. The models (internal/external and quality in use) itself, defined in standard ISO/IEC 9126-1, are accompanied with related technical reports 9126-2 (external metrics), 9126-3 (internal metrics), and 9126-4 (quality in use metrics) [47].

Dromey defines a model framework that places a single level of *quality-carrying properties* between the high-level quality attributes (i.e. the ISO/IEC 9126 characteristics extended with the reusability) and the components of a product [46]. Using the model framework, a quality model can be built per product in bottom-up as well as by top-down approach. The *quality-carrying properties* serve as the intermediaries that link product components to quality characteristics (high-level attributes) and vice versa.

ISO/IEC 25000 [48] suite or Systems and software Quality Requirements and Evaluation (SQuaRE) is the ISO/IEC 9126 successor and is more completed since it is based on previous works and models.

The SQuaRE set of standards, especially ISO/IEC 25010 standard, System and software quality models, due to their maturity and reliance on the previous definition models are a straightforward choice for the definition model.

### 3.1.1 SQuaRE

The International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) both specialized in international standardization, established a joint technical committee, ISO/IEC JTC 1, to deal with standardization in the field of information technology. The international standard series ISO/IEC 250xx Systems and software engineering, Systems and software product Quality Requirements and Evaluation (SQuaRE) were prepared by ISO/IEC JTC 1, Information technology, Subcommittee SC7, Software and system engineering in 2005. A minor revision of SQuaRE series was published in 2014 [48]. The SQuaRE series of standards are depicted in Figure 4.

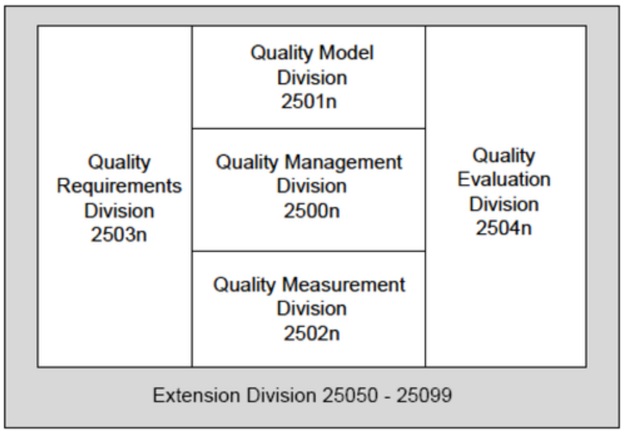
Quality Model Division 2501n
Quality Requirements Division 2503n
Quality Management Division 2500n
Quality Evaluation Division 2504n
Quality Measurement Division 2502n
Extension Division 25050 - 25099

Figure 4 Organization of SQuaRE series of international standards [48]

Software quality models and evaluation were separately covered by the SQuaRE predecessors ISO/IEC 9126 (Software product quality) and ISO/IEC 14598 (Software product evaluation), respectively. The ISO/IEC 14598 was a spin off of the original standard ISO/IEC 9126:1991.

The general goal of creating the SQuaRE set of standards was to coherently cover two main processes:

- Software quality requirements specification,
- Systems and software quality evaluation,

supported by a systems and software quality measurement process. SQuaRE establishes criteria for specification, measurement and evaluation of systems and software product quality requirements.

In order to facilitate those goals and align the customer definitions of quality with attributes of the development process, the SQuaRE includes a two-part quality model (internal/external product quality and quality in use). The ISO/IEC internal/external product quality model i.e. a definition model is shown in Figure 5. The model has eight main characteristics and 31 subcharacteristics hierarchically organized.

The non-functional, ISO/IEC 25010 quality characteristics that have been broadly addressed by the software engineering practitioners and scientists are reliability and maintainability, but as common concepts with a slightly different definitions and without breaking down into subcharacteristics.

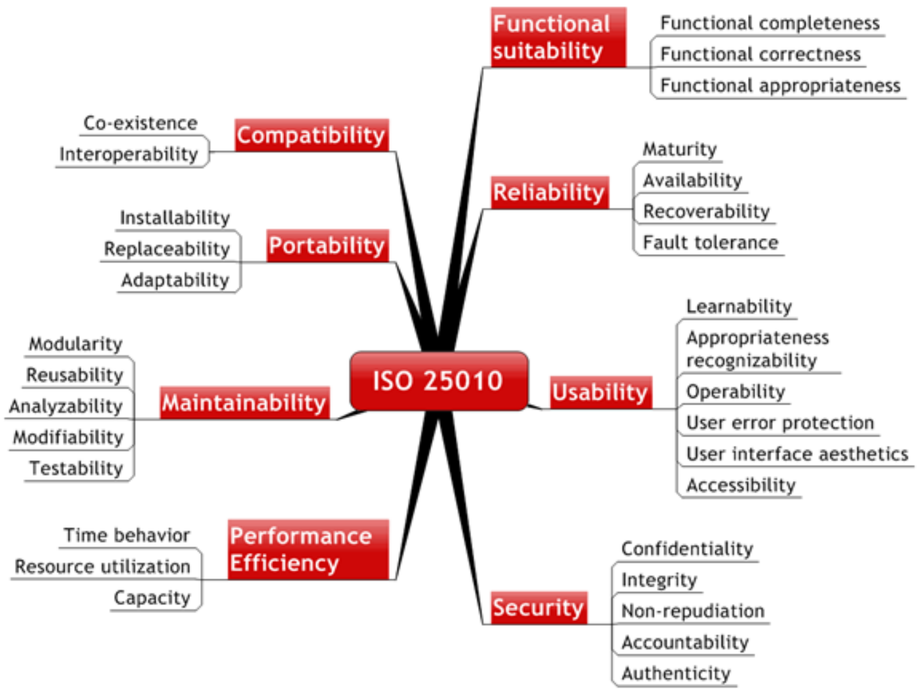

Figure 5 ISO/IEC 15010 product quality model

The ISO/IEC 25010 quality in use model is shown in Figure 6. The model has four characteristics broken down into nine subcharacteristics.

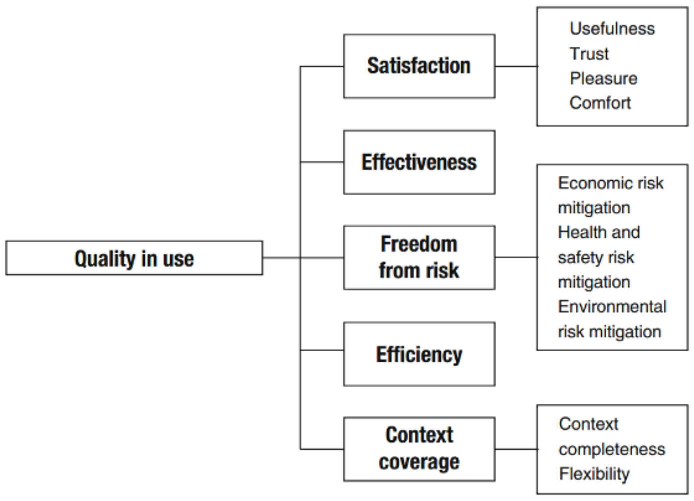

Figure 6 ISO/IEC 25010 Quality in use model

The quality in use model has not been extensively explored by software engineering scientists as the software product quality, but with the actual cloud business models and software-as-a-service (SaaS) concept quality in use is actually very relevant and important. The quality in use model can serve as a definition model for quality user view approach [5] and support trends that software must fulfil end user expectations.

A close term to *quality in use* is user experience. With the SaaS concept, the user experience becomes quality of service (a concept borrowed from telecommunications domain and originally related to telephony service expressed through jitter, delay, et.). Parmakson argued that despite differences between software quality models and quality of service models, there is room for substantial level of alignment between these two models [49]. Since SaaS becomes an important business model, further alignment of software models (mainly software in use model, but also internal/external quality models) with quality of service models is necessity.

QoS relies on the customer's perception and it is not an inherent quality of the whole system representing a service. Several models for QoA are in usage:

- SERVQUAL
- RATER model
- E-SERVICE QUALITY

The main dimensions of service quality are:

- Reliability
- Responsiveness
- Assurance
- Empathy

Relation between ISO/IEC 25010 internal and external software product quality models and quality in use models along the software product life cycles is depicted in Figure 7.

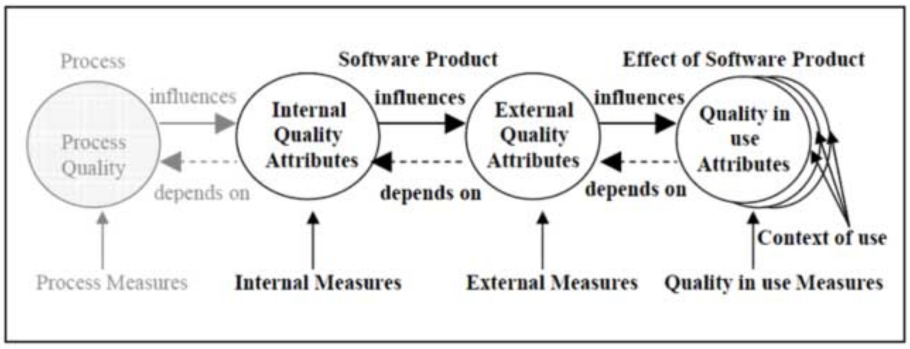

Figure 7 Quality in the lifecycle (Figure C.2-ISO 25010) [39]

Castillo brought into focus the importance of the non-functional aspects and proposed a functional model expressed in UML, REASQ (Requirements, Aspects and Software Quality) [50].

Software quality taxonomy itself is a key factor in ensuring adequate quality. Linking high levels of end user's quality perception, software engineers (developers and maintainers) and software business notions of quality to software systems and products characteristics are necessary to quantitatively asses fulfilment of software quality requirements. Since the whole process involves different stakeholders it is also important to facilitate communication and understanding between them. Widely accepted and validated software measures are necessity for that [48].

The SQuaRE series addresses systems and software product quality requirements specification, measurement and evaluation, while quality management of software development processes is a distinct and separate issue defined in the ISO 9000 family of standards. The main SQuaRE parts that can be used in a general software quality model are the terms and definition, new general reference model and systems product quality (not supported by ISO/IEC 9126). The ISO/IEC 25010 as a software quality definition model defines the software quality taxonomy i.e. comprehensive expressions and terms, and simple and accurate definitions. The main disadvantages are generality and lack of implementation details, as well as the division between quality characteristics and subcharacteristics are sometimes fuzzy and overlapping.

Existing quality definition models lack clear decomposition of complex software characteristics. Quality characteristics and attributes are mostly too abstract to be straightforwardly checkable in a real software system. Moreover, quality definition models, in measurements and metrics accompanying the quality characteristics, often propose some activity measurements that are hardly feasible and in an industry environment cannot be justified [1].

## 3.2 Assessment quality modelling

Quality assessment models often extend quality definition models, usually focusing on one or a few software quality characteristics of choice, to control conformance to requirements. Assessment models can be used to objectively specify and control stated quality requirements during *software requirements analysis* (*software implementation* core process) and *software verification* (*software support*) processes. During *software implementations*, the software quality assessment model is the basis for all quality measuring including the product, processes/activities and the environment [34]. Besides using quality assessment models during the *software implementation* process or phase, they can furthermore be used to define the criterion for quality certifications [4].

Software metrics based models are commonly used to assess the quality of a given system [4]. A number of metrics for software quality measurements have been proposed, but fail to clearly explain the impact and relation that specific metrics have on software quality and specific quality characteristic [7]. The aggregation of metric values in bottom-up fashion along the hierarchical levels is also challenging due to the lack of a clear semantic [4].

The following requirements to be met by a practical model based on source code analysis are defined [1, 51]:

- Language, architecture and technology agnostic measures,
- Measures availability and easiness of collecting
- Straightforward definition provides easy computation
- Simple measures, understandable to non-technical staff and management.

- Root-cause analysis should be enabled by clear causality between code-level properties and system-level quality.

The overview of software quality assessment models will be started with some basic metrics, and then followed by some common metrics aggregations and quality assessment models.

### 3.2.1 Software metrics

Software metrics deal with quantification of software properties and characteristics (mainly by measurements of source code), and aspects of *software quality testing* and *assurance* such as recording and monitoring defects during development and testing phases. Software metrics should be a cornerstone in an ultimately empirical discipline such as software engineering, but they are often misunderstood and misused (Fenton 2000). Software metrics seldom measure the intrinsic quality attributes, but they pretty well compare related quality attributes of different parts of a software systems. In essence, software metrics is more a modelling than a measuring tool [52]. Moreover, software measurement or metrics theory or scientific research is out of step with practice or industrial application contribution [1]. The single biggest boost for industrial metrics in the US was the CMM (Capability Maturity Model, the CMMI predecessor), since evidence of use metrics is important for achieving higher levels of CMM [31].

The key basic metrics, chronologically ordered, are:

- Lines of Code (LOC) or KLOC for thousands of line of code, used as a surrogate measure of product size based on assumption that size is critical for both quality and effort/cost.
- Software complexity such as cyclomatic complexity (McCabe 1976) or Halstead model [53].
- Functional size such as Albrecht function points, supposing to be programming language agnostic.

Metrics in an industrial setting are also well studied but mainly on private data sets [54].

Fenton and Neil divide software metrics into two components, the one defining the actual measures, and the other one concerned with collecting, managing and using the measures [31]. For the latter component, an idea that metrics activities shall be goal-driven i.e. Goal-Question Metric (GQM) was a real breakthrough [55]. Both quality model design methods, GQM and McCall's factor-criteria-metrics (FCM) recommend combination of different metrics in order to fully assess the goal or factor i.e. the software quality characteristic or subcharacteristic. Moreover, we are looking for insights in the quality of the whole system based on the metric values of its low-level components such as classes and methods. Consequently, the two big challenges for software metrics assessment practice are [56]:

- Metrics integration or combining different software metrics;
- Aggregation of software metrics (an individual or a composes) defined per components into one high level value.

#### 3.2.1.1 Metric composition

Metric composition is a necessity for a full and reliable assessment of software characteristics or subcharacteristics. *Changeability* was subcharacteristics of *maintainability* characteristic in ISO/IEC 9126 with definition: "the capability of the software product to enable a specified modification to be implemented". These subcharacteristics may be addressed by several metrics, such as number of source lines of code (LOC), cyclomatic complexity, number of methods per class, and inheritance depth [56]. ISO/IEC 25010 have changed decomposition of *maintainability* characteristic and define new *modifiability* subcharacteristics as combination of old, ISO/IEC 9126, *changeability* and *stability* subcharacteristics. That additionally emphasizes the metrics combination challenge. Main challenges for metrics composition are the ranges of individual metrics and different meanings. That may impose a usage of specific composition method for each characteristic.

#### 3.2.1.2 Metric aggregation

Common metric aggregation techniques are aggregation by simple or weighted averaging. Averaging the results of a metric, especially the simple one, has an undesirable smoothing effect that can dilute bad results in the overall acceptable quality. More recently, there is an interdisciplinary trend in scientific literature on aggregation techniques for software metrics in using more advanced aggregation techniques well known in econometrics for their applicability to studying income inequality, such as the Gini coefficient, and the Theil and Hoover indices [51, 52, 57, 58].

### 3.2.2 Assessment models

Quality assessment models often assess specific quality characteristic and they are relatively simple. There are also integrated, comprehensive quality assessment models that assess overall software quality, but they much more complex.

Software maintainability and its impact on software industry is one of the most important aspects of software quality due to the related incurring costs. The total cost of maintenance is estimated to 40% of the development cost, or even worse Hewlett-Packard estimation from 1992, that 40 to 60 percent of the cost of production is maintenance expenses [52]. Although ISO/IEC 9126 recognizes maintainability as one of main software product quality characteristics, as well as ISO/IEC 25010, they do not provide measures for estimating maintainability on the basis of a source code. The proposed metrics for assessing the maintainability are based on the performance of the maintenance activity by the technical staff [51]. ISO/IEC 9126 decomposes maintainability into analysability, changeability, stability, testability and maintainability conformance subcharacteristics, while ISO/IEC 25010 keeps analysability and testability, changeability and stability are replaced with modifiability, add reusability and modularity while discards maintainability conformance.

Using regression analysis upon the measurements of source code and calibrating these results, the Maintainability Index (MI) has been proposed (Oman 1994) (Coleman 1994). The MI is a composite index, based on several different metrics for a software system:

$$171-5.2\ln(HV)-0.23CC-16.2\ln(LOC)+50.0\sin\sqrt{2.46*COM}$$

where:

*HV* is the Halstead Volume metric, a composite metric based on the number of (distinct) operators and operands in source code;
*CC* is the Cyclomatic Complexity metric;
*LOC* is the average number of lines of code per module;
*COM* is optionally the percentage of comment lines per module.

The higher the MI, the more maintainable a system is supposed to be. The original MI defined an assessment whose relation to a definition of quality was unclear. The main problems with the MI identified by (Heitlager 2007) are:

- Since the MI fitting function is obtained by statistical correlations, there may be no causal relation between the measured values or metrics and the MI value derived from them.
- The average Cyclomatic Complexity hides the presence of high-risk modules due to a power law distribution of complexity per module.
- The Halstead Volume metric is difficult to define and compute.
- From the MI value it cannot be concluded which subcharacteristics of maintainability contribute to that value.

Heitlager alternated the original MI and chose some source code measures mapped via source code properties onto the subcharacteristics of maintainability according to ISO/IEC 9126 (Fig. 8), [51]. Thus he built a software maintainability assessment model using the elements of a definition model (ISO/IEC 9216).

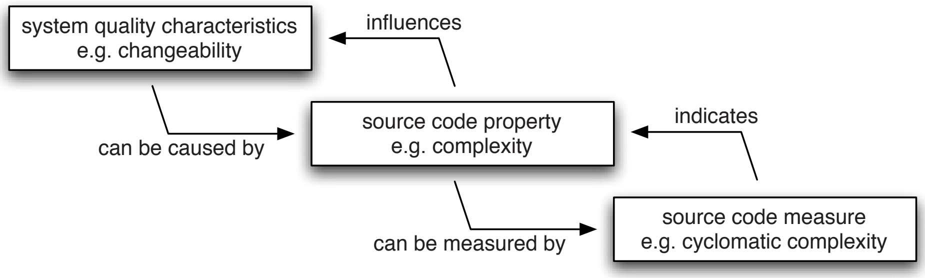

Figure 8 Mapping of maintainability subcharacteristics to source code measures

Overall or integrated quality assessment model represents all or a big subset of quality characteristics and relationships that affect software quality. This model requires more data and it is too complex to represent by an algorithm. Some definitions of measurement of quality characteristics and subcharacteristics are based on subjective assessment from project and quality assurance group members. Those subjective remarks could come with significant deviations.

## 3.3 Prediction models

The common model will surely help in communication, assessment and improving overall software quality. Ideal prediction model is the most advanced quality model because it can also be used for the definition and assessment of quality. By such approach we will get a model appropriate for different purposes such as:

- Requirement management
- Resource estimation
- Quality estimation

Project management will benefit a lot from the above estimation. Software defect predictors serve as identification of fault-prone software modules to properly allocate resources for defect detection and removal [59, 60]. Both references validated their claims on large telecommunications software systems (C++ and Java programming languages). Reliability predictors help in release planning or *when to stop testing* [4]. Coleman proposed the use of automated maintainability assessment for different management decisions such as *buy-versus-build* decisions, test resource allocation and the prediction of defect-prone components [52].

An important class of software quality prediction models is reliability growth models [4].

### 3.3.1 Software Reliability Models

Reliability has been considered as the most important software quality characteristic [61] and consequently attracted lots of interest and research in the software community, especially software reliability modelling. ISO/IEC 25010 lists reliability as one of the eight main characteristics of software product quality, with subcharacteristics:

- Maturity
- Availability
- Fault tolerance
- Recoverability.

The ability to provide evidence or trustable prediction of reliability is a prerequisite for the new software release acceptance in business and safety critical applications. Majority of existing software reliability models are based on slightly different software reliability understanding, definitions and data, hence cannot be explicitly linked to a definition of quality as it was described by the ISO/IEC 25010 standard. ISO/IEC 25010 defines reliability as a degree to which the software product can maintain a specified level of performance when used under specified conditions. Zeljković used ANSI/IEEE definition of reliability as the probability of failure free software operation for a specified period of time in a specified environment [62]. Faqih listed reliability and availability as important but separated software quality attributes, while Wood sees software reliability is a critical component of computer system availability [35, 63].

Ideally a software reliability model should measure or estimate ISO/IEC reliability subcharacteristics and then compose them into reliability software quality characteristic.

Su et al. classified software reliability models into two major categories: deterministic and probabilistic (stochastic) [64]. The deterministic one deals with the number of distinct operators and operands in the program. The probabilistic one is used to study the failure occurrences and the fault removals as probabilistic events i.e. accounts for random error. The probabilistic models can be further broken down into different classes such as error seeding, failure rate and non-homogeneous Poisson Process (NHPP).

The error seeding software reliability model originated with fish tagging. Fishery biologists tagged some fish to estimate the size of fish population in a fishery area. The ratio of the re-caught tagged to the total number of tagged fish is assumed to be the same as the ratio of the total fish caught to the total number of fish in the area. Similarly, error seeding is inserting deliberately errors into software. Alternatively the errors found by debugging can be tagged. Then, the number of indigenous errors in the software can be estimated based on the number of errors found by a debugger unaware of the seeded/tagged errors and the number of errors appearing in both sets. Neufelder strongly advised against error seeding implementation [65].

The NHPP model, as a representative of time domain, non-homogenous Markov models (Figure 9), is the most popular model in literature due to the assumption that it has the ability to well describe the software failure phenomenon [66]. The number of faults present in a software product is a random variable assumed to display the behaviour of a non-homogenous Poisson Process. The first NHPP model was proposed by Goel and Okumoto [38]. The Goel-Okumoto model is an exponential model with a concave characteristic curve.

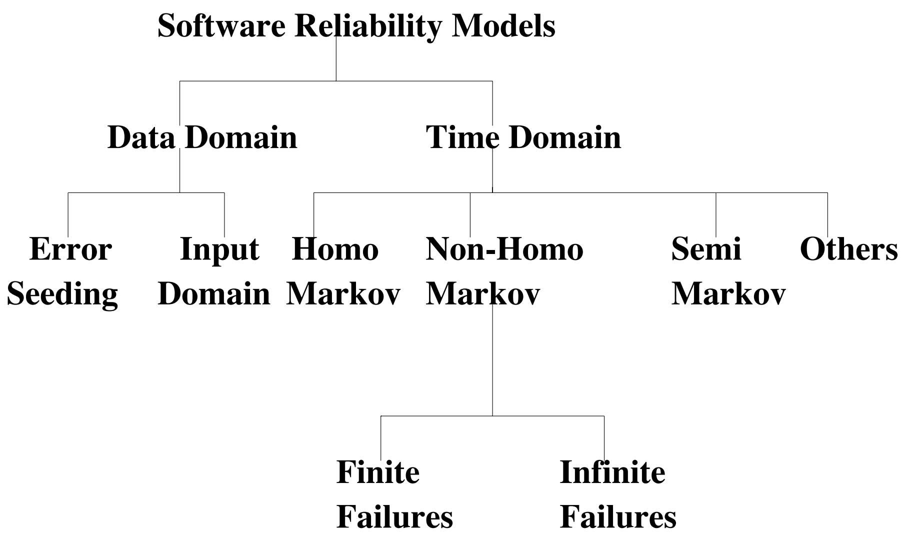

Figure 9 Classification of Software Reliability Models [66]

Another important classification is the distribution of the number of the failures per time such as the Poisson and binomial distribution, members of the large class of exponential families of distributions. The development of all these models was based upon concepts adapted from hardware reliability theory [61].

Wood classified software reliability models in two categories based on used data: models that predict reliability from design parameters and models attempt to predict reliability from test data [35]. The first type of models use code properties and measures such as lines of codes, nesting loops, external references, input/output operations to predict the number of defects in software and they are usually referenced as *defect density* models. The type of models relied on test data are called *software reliability growth models* and they are trying to *statistically correlate defect detection data with known functions such as an exponential function* in order to estimate future behaviour [35].

A similar classification on static and dynamic reliability models is proposed by [67]. The static software reliability models are based on source code properties, while the dynamic models are based on *temporary behaviour of debugging process during test phase*. Models describe error detection are called Software Reliability Growth Models (SRGMs).

#### 3.3.1.1 Software reliability growth models

Software reliability modelling appeared around the early 1970s with basic approach of modelling defect data shown at testing to predict future behaviour in two main classes: failures per time period and time between software failures. SRGMs can give us indication of software readiness for release as well as indication of the number of failures in operation after software release.

Most of SRGMs have the total number of defects contained in a set of code as a cornerstone parameter. With the total number of defects and the current number of discovered defects,  we can get the residual defects, which can lead to the failures during the software operation.

The next SRGM ground parameter is functional relation of number of defects and time. During the test the time is represented by amount of testing and can be expressed as calendar time, execution time or number of test cases. The defect detection rate is reciprocal of the time between detected defects. Software defects usually lead to software failures, but defects can exist even if the software continues to operate. While the cumulative number of defects increases, the defect discovery rate decreases as the amount of testing increase.

Due to difficulties to extrapolate system operation time from the time representation during the testing, it is consequently difficult to extrapolate failure rate during the system operation from the defect discovery rate during the testing. Thus the focus is on number of remaining defects in a code instead the failure rate. The residual defects is an upper limit of failures a user could encounter during the operation of software [35].

Defect detection data is statistically interpolated by mathematical functions, which are used to predict future failure rates or the residual defects in software. Two shapes of software reliability growth model functions are common: concave and S-type, Figure 10.

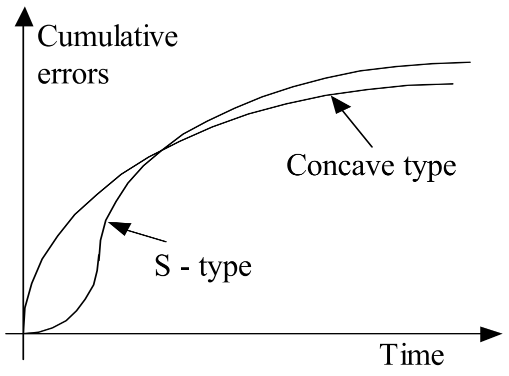

**Figure 10 Common shapes of the SRGMs**

The common software reliability growth model functions are [35, 62]:

- Goel-Okumoto $a(1 - e^{-bt})$
- Gompertz S-type $a(b^{c^t})$
- Weibull concave $a(1 - e^{-bt^c})$
- Pareto concave $a[1 - (1 + \frac{t}{\beta})^{1-\alpha}]$
- Yamada Raleigh S-type $a\left\{1 - exp\left[-r\alpha\left(1 - e^{-\frac{1}{2}\beta t^2}\right)\right]\right\}$

Those functions should be customized in order to fit sample defects data i.e. the function parameters should be determined. The basic methods for estimating the parameters of a statistical model are:

- Maximum likelihood estimation (MLE)
- Least square estimation (LSE).

The proposed software reliability growth functions are nonlinear and the MLE and LSE are not optimal. Different methods for parameters optimization are proposed such as particle swarm optimization and ant colony optimization [67, 68].

Zeljković claims that software reliability cannot be calculated during the design phase [62]. But Goel-Okumoto model,

$$m(t) = \alpha\left[1 - (1 + \beta t)e^{-\beta t}\right]$$

for which the $\alpha$ and $\beta$ parameters are estimated using maximum likelihood estimation method and real software system failures data, shows good matching between model and real complex software system.

Faqih brings some review of software reliability growth model criticisms [63]. The ground for the criticisms is that authors often unjustifiably make some assumptions in their mathematical formulation of the model to provide mathematical tractability. For example, NHPP models assume the nature of a software faults and the stochastic behaviour of a software failure process. Contrary, the neural network build a model adaptively from the given failure data [64]. Moreover, most of the proposed software reliability models are not tested and validated by using real data. The main reason is that software companies are unwilling to share their data on software defects and failures. Software reliability growth models are based on hardware reliability growth models, which calls for additional caution because software cannot be worn out. Another limitation of the SRGMs is that they can be applied from integrated testing onward (not for the early phases of the life cycle).

# 4 Software quality managerial decision support

The most important requirement of software metrics in general is to provide reliable information to support quantitative managerial decision-making during the whole software lifecycle span [31]. This chapter will try to roughly sketch all needed requirements and components for industrial, integral and causal modelling of software quality for a support system/tool for managerial decisions.

The universal and working software quality definition/assessment/prediction model that crosscuts different usages and applications related to software quality will be very convenient and useful due to its eventual familiarity within software community but the big open question is "Is such model plausible?".

The quality software engineers traditionally measure certain metrics. Fenton as traditional software industry metrics lists size (LOC or similar metric), defect counts and efforts in person-moths due to its clear meaning and easy collecting [1]. Chang sees issue with mapping of a decision maker's perception to exact number produced by software quality models [69].

Some methodologies and best practices were proposed for building software quality models:

- An ideal software quality model should embrace a definition model or taxonomy, an assessment model (including metrics), and some prediction ability [4]
- Goal-question-metric, GQM [55] paradigm.

Those approaches are overlapping. The GQM methodology proposes top down approach. First the goals must be specified according to the organization and project needs, then those goals must be traced all the way down to the data that are going to define those goals operationally, and finally a framework for interpreting the data with respect to the stated goals must be provided. Furthermore, in fitting a model to a given data set, the model's assumptions must be respected. For example if a selected model (e.g. Scheneidewind's model) assumes that the time intervals over which the software is observed or tested are all the same magnitude, it cannot be used with data for which the assumption has not been fulfilled. For reliable future predictions, the environment in which the data have been collected must not change considerably from one in which the software is being tested or observed [61].

A goal could be defined for a product, process or resource with respect to various quality models, points of view and environments. Our goal is quantitative support for managerial decision-making emphasizing software quality issues.

The appropriate quality model should embrace everything at least as a constant. In order that a fixed parameter (constant) could be replaced by a model, some hooks and interfaces toward the dependent values in the rest of model must be defined. The process could be replaced by a constant e.g. waterfall or agile process. Resources, for simplicity reason, could also be fixed and could be represented by costs/hour constant.

So our main point of interest will be software product during its whole life cycle. Software product could be further decomposed via costs, time and quality attributes/dimensions. Cost and time are pretty clear dimensions, while quality is still an elusive concept [7].

The managerial quantitative support includes trading off and balancing software in optimal time/cost/quality space. For that goal, software quality must be modelled in order to the define manoeuvring space. Wagner proposes representation of the 3D space of quality, time and costs by one-dimensional space of cost or profit, because profit is in the focus of commercial software [16]. By choosing costs as universal measure for different software quality aspects/activities we cover value based quality view [5]. A quality model is our main point of interest. It must clearly define quality codomain in order to know what costs and time degrees of freedom are.

Concerning the points of view, the dominant view for software quality has been product view [5], but for a contemporary, integral and trustable quality model user view must be included, either through user related software product quality model main characteristic of usability or *software in use* model.

Concerning the environment, the main area of interest will be telecommunication software domain, due to availability of data and comprehensive knowledge on software *implementation*, *maintenance* and *quality assurance* processes. The telecommunications (control) software as well as other parts such as hardware is planned in advance through different roadmaps. Those roadmaps are evolutionary, mutually dependent e.g. a major software release has to follow an introduction of new hardware. Moreover, the roadmaps are related to or have to implement the company's business strategies, customer requirements, user expectations, regulation frameworks and generally advancements in telecommunications e.g. transition to the next generation. Within those roadmaps there are many important decisions, such as selection of software base line for the next major release, selection of software units and subsystems for redesign (after "stinker" analysis), testbed planning, release dates, all significantly dependent on software quality. A general and simple model of quality, constantly updated through time and different software life cycles phases and processes, will be invaluable tool for business (operational and strategic) planning and decision making.

A usable software quality model for managerial decision support system in telecommunication (control) software industry should satisfy majority of objectives and constraints that are reasonable to implement, should be cost-effective and should provide feedbacks to software development processes and impact analysis [2]. The common approach is to extract desired software systems attributes/characteristics from a chosen taxonomy (or software quality model) that optimally fits to the requirements, domain, software system purpose and end users. [70] chosen functionality, usability, reliability and efficiency from the ISO/IEC 9126 quality model and Bayesian Belief Network for their E-commerce systems model concerning the quality.

There is an abundance of methods that deal with the various software quality aspects, but most of them are related to some niche of product-based view, such as maintainability and reliability. On the other hand, most common practices in software industry deal with software quality related to the functional requirements, while the non-functional requirements are usually neglected. Gupta et al. classified techniques for building quality models into algorithmic, such as regression analysis, and non-algorithmic, such as probabilistic and soft computing [33]. The quality models building techniques are also compared in terms of the following modelling capabilities:

- Explaining outputs,
- Suitability for small data sets,
- Adjustability for new data,
- Visibility of reasoning process,
- Suitability for complex models,
- Inclusion of known facts.

The key difference between the larger and the small data sets is that the larger data sets permit the use of non-linear statistical and neural network models. The conclusions are that the techniques are suited either for classification or prediction quality models and that there is no single technique that fulfils all requirements: flexible, transparent and reusable quality model. Case-based reasoning and fuzzy system modelling techniques are seen as the most promising.

In quality modelling for telecommunications control software development all capabilities listed above are desirable, except suitability for small data sets.

In order to overcame the constraints and prerequisites of SRGMs, some soft computing approaches and techniques are used. Soft computing techniques are a family of problem solving approaches that tackles the imprecise and uncertain real world problems by mimicking the biological problem-solving already existing in nature. Plants, animals and human beings exhibit flexible, adaptive and smart approaches to the real world problems. The main members of the soft-computing techniques are [71]:

- Fuzzy system;
- Neural networks;
- Chaos theory;
- Evolutionary computing;
    - Evolutionary algorithm
    - Swarm intelligence
- Support Vector Machine (SVM);
- Bayesian network.

Soft computing techniques are used in many areas such as machine learning and artificial intelligence applied in engineering.

The soft techniques are often used for assessment or to predict a single notion of software quality or characteristic such as reliability and compete against traditional SRGMs. There are also attempts to model the integral software quality and all or subset of quality characteristics represented by some integrated hierarchy models [72].

Wagner and Deissenboeck identified six different dimensions of software quality modelling [73]:

- Purpose with three main types: Constructive, predictive and assessing [4]
- View corresponding to five different approaches [5]
- Attribute (i.e. software characteristics)
- Phase (could be defined by the phases of the software product life cycle)
- Technique (focus on a single technique e.g. system test)
- Abstractness (level of details).
- Definition models ISO/IEC 9126 as well as 25010 mix criteria from different dimensions.

The quality definition models, at that time ISO/IEC 9126 and latter OSI/IEC 25010, have failed to establish a broadly accepted definition of quality because they mix criteria from different dimensions and the characteristics are not sufficiently described in order to be assessable [73]. The authors recommended to use costs as measure for all quality related attributes and mapping of desired software attributes or characteristics to corresponding activities that could be assessed from economic point of view and associated with some costs.

Moreover, costs certainly can serve as a common denominator for integration of different quality assurance techniques and building a model/framework for holistic approach to software quality during the whole span of software product life cycle.

## 4.1 Software quality economics or cost approach

The value-based approach is certainly very important for industrial software, where economic reasoning directly drives most management decisions. If we look for a managerial software decision support based on software quality, understanding of costs and benefits of software quality is essential. The cost is one of the main parameters of software development process along the quality and time.

Wagner proposes cost as an universal unit including quality due to two primary reasons: most software projects are done by the companies driven by profit and quality as a multifaceted concept needs some common quantization or denominator [16]. The same author overviewed the results of empirical studies about the efficiency of defect-detection techniques and associates them with a model of software quality economics [74]. As software quality assurance with defect-detection techniques is accounted for almost half of development costs, the economics of software quality assurance is highly important in practice. The understanding of the economics is essential for managerial decision on how many testing are enough and if the agreed level of quality is reached. Wagner also tried to optimize the usage of defect-detection techniques (in which order they should be applied and with what effort) [74]:

- Dynamic testing (executes software with the aim to find failures)
- Reviews and inspections
- Static analysis tools (tool-based analysis of source code without executing it).

Quality costs could be attributed to preventing, finding and correcting software defects and failures and one possible classification is given in Figure 11 [74].

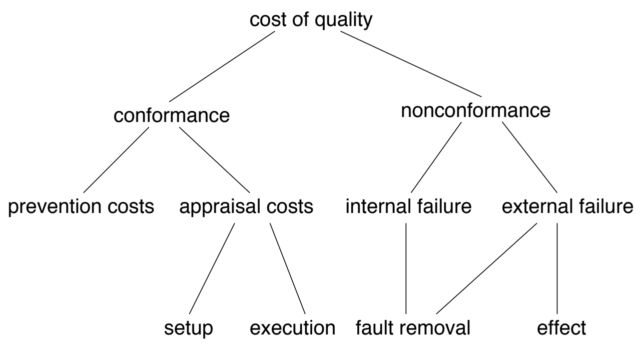

**Figure 11 Overview of the costs related to quality [74]**

The classification depicted in Figure 11 makes a distinction between prevention, appraisal and failures costs and hence it is called PAF model. The model is focused on reliability and as such is not completed and exhausted e.g. there is no maintenance costs.

The Wagner's quality economic model has three main costs: direct costs, future costs and revenues or saved costs. Direct costs are directly related to the application of the defect-detection techniques, the future costs contain the incurred costs while revenues comprise saved costs.

The direct costs of defect-detection techniques *A* are defined as:

$$d_A = u_A + e_A(\mathrm{t}) + \sum_{\mathrm{i}}(1 - \theta_A(\mathrm{i}, \mathrm{t}_A))\mathrm{v}_\mathrm{A}(\mathrm{i}),$$

where $\mathrm{u}_\mathrm{A}$ are the setup costs, $\mathrm{e}_\mathrm{A}(\mathrm{t})$ are the execution costs of t long application, $\theta_\mathrm{A}(\mathrm{i}, \mathrm{t})$ is the probability that defect-detection technique A does not detect error i when applied with effort $\mathrm{t}_\mathrm{A}$ and $\mathrm{v}_\mathrm{A}(\mathrm{i})$ is the error removal costs specific to technique A.

A metric used direct cost of defect-detection technique is the return on investment:

$$ROI = \frac{r_X - d_X - o_X}{d_X + o_X},$$

describing the ratio between revenue decreased for costs and costs.

For practical purposes the model is simplified and errors/faults are categorized in defect types, and defects types have specific distributions regarding their detection difficulty, removal cost and failure probability.

## 4.2 Bayesian networks

Bayesian networks (BNs) are a modelling technique for causal relationship based on Bayesian inference. They are also known as Bayesian beliefs networks (BBNs) or just belief networks.

Basically, Bayesian inference mathematically describes how to change existing beliefs with appearance of new data. It makes possible to combine old knowledge with the new data. According to Bayes, observations should be considered as dynamical influence on judgment. Bayes inference could be drawn from the joint probability of events *A* and *B*, $P(A \wedge B)$:

$$P(A \wedge B)=P(B \wedge A),$$
$$P(A|B)P(B) = P(B|A)P(A),$$
$$P(A|B) = \frac{P(B|A)P(A)}{P(B)},$$

where are:

*P(A)* probability of hypothesis *A* (prior),
*P(B)* probability of new event *B* (evidence),
$P(A|B)$ joint probability i.e. probability of *A* if *B* occurs (posterior),
$P(B|A)$ joint probability of *B* after event *A* (likelihood).

Denominator *P(B)* could be further broken down as

$$P(B) = P(B|A)P(A) + P(B|\sim A)P(\sim A)$$

and Bayes inference could be written as

$$P(A|B) = \frac{P(B|A)P(A)}{P(B|A)P(A) + P(B|\sim A)P(\sim A)},$$

or more descripting as

$$posterior = \frac{likelihood \times prior}{evidence}.$$

In continuous domain the Bayes decision rule is:

$$P(\omega_i|\bar{x}) = \frac{p(\bar{x}|\omega_i)P(\omega_i)}{p(\bar{x})},$$

where:

$\bar{x}$ is a feature vector in $\Re^d$ domain,

$\omega_i$ is a finite set of $c$ possible states $\{\omega_1, \dots, \omega_c\}$,

$p$ is a density probability function with $\int_{-\infty}^{+\infty} p(x)dx = 1$,

$P$ is a probability within the range $[0,1]$ with $P(X) = \sum_{i=1}^{c} P(x_i) = 1$.

$P(\omega_i)$ is an assumption or hypothesis that state $\omega_i$ will occur i.e. existing system model. $p(\bar{x} \mid \omega_i)$ is probability of state $\omega_i$ after a new, generally known evidence $\bar{x}$ occurs. We are looking for the latest probability of state $\omega_i$, $P(\omega_i \mid \bar{x})$, after new evidence $\bar{x}$. $p(\bar{x})$ is usually called scaling factor.

The Bayesian network could be graphically represented as a directed acyclic graph where vertices representing uncertain variables and edges representing directed relationships between variables, Figure 12.

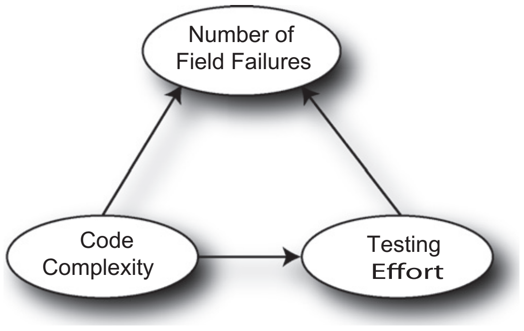

Figure 12 A simple Bayesian network [75]

Each vertex (node or variable) has a node probability table (NPT) that defines the relationships and uncertainty for the variable. An example of a NPT is shown in Figure 13.

| Testing effort | Low | | | High | | |
|---|---|---|---|---|---|---|
| Code complexity | Low | Med. | High | Low | Med. | High |
| <100 | 0.4 | 0.3 | 0.2 | 0.6 | 0.55 | 0.5 |
| >100 | 0.6 | 0.7 | 0.8 | 0.4 | 0.45 | 0.5 |

**Figure 13 An NPT for variable *Number of field failures* [75]**

The variables are usually discrete with a fixed number of states e.g. variable *Number of field failures* has two states: below 100 and above 100. For each state the NPT gives probability that the variable is in this state, but different for each influence of existing parent nodes. In this example the probability that the variable *Number of field failures* is below *100* failures is 0.2 where parent variables *Testing effort* is *Low* and *Code complexity* is *High*.

The process of building a Bayesian network comprises identification of important variables, representing them as nodes and connecting in a acyclic directed graph respecting all mutual causalities, and specifying the node probability tables (NPTs).

Fenton and Neil modelled software defects insertion and detection process (Figure 14), [31, 76].

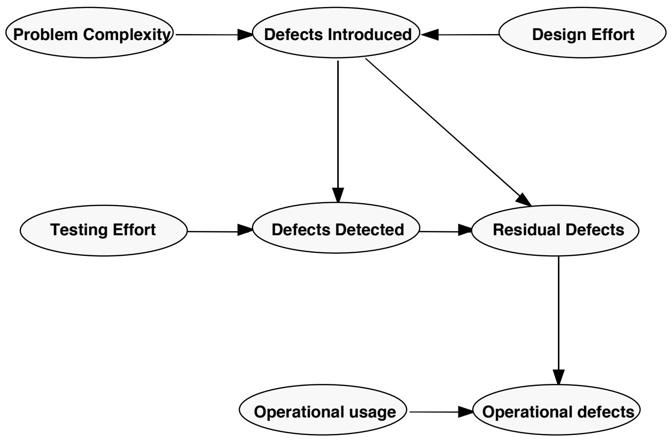

**Figure 14 Simplified Bayesian network for predicting pre-release defects and post-release failures [31]**

The Bayesian network depicted in Figure 14 and related node probability tables are based on a mixture of empirical data and expert judgements. The approach using Bayesian networks besides dealing with causality and uncertainty also allows introducing and combining different, often subjective and elusive, evidences.

The BN approach is also applied to resource modelling and prediction [76], Figure 15.

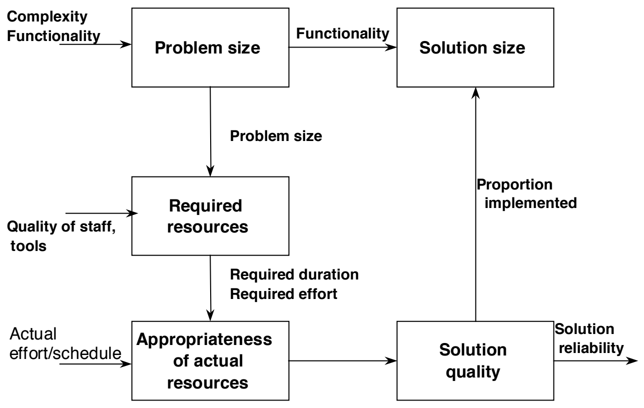

Figure 15 Causal BN model for software resources [76]

The boxes in Figure 15 represent actually subnets. Each of the subnets contains variables e.g. *problem size* subnet comprises complexity and functionality while subnet appropriateness of actual resources comprise required duration and efforts, actual effort and duration. The latter subnet is important and distinguishing because it through cause-effect relationships challenges the *required resources* as a best practice estimation by actual efforts and schedule.

Fenton brings more general approach that allows BNs to predict defects along the whole software lifecycle [77]. Previous approach required different and customized BNs for each process due to availability of different metrics. Marquez introduced hybrid BNs containing both discrete and continuous variables for reliability modeling [78].

Software quality or at least some characteristics such as reliability and maintainability [79] may be justifiably predicted by a Bayesian network due to its ability to combine many different sources of evidences such as test result and process information. Such a BBN could also be useful for examination of various trade-offs, but will not be sufficient for managerial decision reliable support. [31, 80] recommended multi criteria decision aid/making (MCDA/M) for inclusion of other relevant criteria concerning software product such as time, cost and company policy.

## 4.3 Multi Criteria Decision Making

Multiple-criteria decision analysis (MCDA) or multiple-criteria decision-making (MCDM) is a part of operations research discipline that explicitly considers multiple criteria in decision-making environments. Some criteria are usually in conflict e.g. cost and quality, and to bring more informed and qualified decision, complex problem shall be structured well and multiple criteria explicitly considered. One of the main efforts of MCDM research is finding a way of trading off between criteria.

The modern multiple-criteria decision-making discipline started in the early 1960s. There are different MCDM problems and methods. The main classification of MCDM problems is based on solution definitions:

- Multiple-criteria evaluation problem: a problem consists of a finite numbers of solutions known in advance. The problem is either to find the best alternative or to sort solutions.
- Multiple-criteria design problem: solutions are not explicitly known. A solution can be found by solving a mathematical model.

Different approaches and methods have been developed for solving MCDM problems of both types. The principal MCDM approach is multi-attribute utility theory (MAUT). Another popular approaches are [81, 82]:

- Fuzzy sets to model and solve fuzzy problems
- Evolutionary Multi-Objective Optimization (EMO)
- Analytic Hierarchy Process (AHP)

Since the introduction of MCDM, a variety of methods have been introduced for application in different disciplines such as politics, business and environment. Applications of multiple-criteria decision-making have started to be considered also for applications in software engineering [83]. Stamelos and Tsoukias used a multi-criteria model to evaluate the quality of software [84]. Ruhe et al. prioritize software requirements by use of the Analytical Hierarchy Process (AHP) [3]. Chang et al. propose Fuzzy Analytic Hierarchy Process (FAHP) combining the Analytic Hierarchy

Process (AHP) multiple-criteria decision-making method for resolving and fuzzy theory [69].

Kornyshova proposes use of multi-criteria methods in software engineering in three steps [80]:

- Structuring specific decision-making situation,
- Considering decision-making situation specificity,
- Application of multi-criteria method adapted to this concrete situation.

# 5 Conclusion and future work

The foreseen goal is a managerial decision support system with focus on software quality. Although the emphasis will be on software quality, all related software engineering aspects are analysed in order to come closer to a trustable managerial support system. Such system should tackle the whole software life cycle, using also the historic data, to get some solid indicators as soon as possible and to leave space and time for using feedback for improvements and corrections of software development process. The software quality itself should be modelled by following the goal-question-metric (GCM) approach [55]. Another requirement on the software quality model is causality relationships between metric and final goals, which will provide root cause analysis and feedbacks for managerial decisions and actions on source code ground. The software quality model is also constrained by mathematical or statistical ground. Other constraints are industry and particularly telecommunication control software industry best practices, already existing metrics and measurements and its collection, applied development process (these days mostly agile and Scrum development processes) and software development project management.

The appropriate software quality model or more models should be built. A sound choice for the start is a software definition model i.e. a common taxonomy that will facilitate communication and negotiations on software quality issue. The ISO/IEC 25010 looks as best choice because it is an international standard accepted by the all important standardization organization, it is new and based on the previous, hierarchical software quality models. The ISO/IEC 25010 quality definition model main quality characteristics and sub-characteristics are overlapping and define software quality in different dimensions [73], so the next step could be selection of quality characteristics that are orthogonal to some extent and can catch the product, user, manufacturing and value notions of software quality [5]. The selection of such characteristics could start with reliability, maintainability and usability characteristics. The selected characteristics will drive selection of appropriate metrics. Above such definition model, an assessment and prediction model layers can be built. The prediction power of model can be facilitated by causal modelling and deploying of Bayesian Belief Networks that could deal with uncertainties.

To put everything in a broader context of a managerial decision support system for software development, costs could serve as a common measure for different properties and characteristics of software quality and multiple-criteria decision-making could serve as a high-level integration framework. Such theoretical and scientifically grounded model could then be tailored and customized and verified by using real telecommunication industry data as a practical managerial decision support system for management of software projects in telecommunications domain.

## References

[1] N. E. Fenton and M. Neil, "Software metrics: successes, failures and new directions," *Journal of Systems and Software,* vol. 47, pp. 149-157, 1999.

[2] G. Ruhe, "Software Engineering Decision Support–Methodology and Applications," *Innovations in decision support systems,* vol. 3, pp. 143-174, 2003.

[3] G. Ruhe, A. Eberlein, and D. Pfahl, "Quantitative WinWin: a new method for decision support in requirements negotiation," in *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, 2002, pp. 159-166.

[4] F. Deissenboeck, E. Juergens, K. Lochmann, and S. Wagner, "Software quality models: Purposes, usage scenarios and requirements," in *Software Quality, 2009. WOSQ'09. ICSE Workshop on*, 2009, pp. 9-14.

[5] D. A. Garvin, "What does product quality really mean," *Sloan management review,* vol. 26, 1984.

[6] V. A. Shekhovtsov, "On the evolution of quality conceptualization techniques," in *The evolution of conceptual modeling*, 2011, pp. 117-136.

[7] B. Kitchenham and S. L. Pfleeger, "Software quality: The elusive target," *IEEE software,* vol. 13, pp. 12-21, 1996.

[8] S. GROUP, "The CHAOS Manifesto–Think Big, Act Small, last accessed on 27 June, 2014," 2013.

[9] C. Cachero, C. Calero, and G. Poels, "Metamodeling the quality of the web development process' intermediate artifacts," in *Web Engineering*, ed: Springer, 2007, pp. 74-89.

[10] R. C. De Boer and H. Van Vliet, "QuOnt: an ontology for the reuse of quality criteria," in *Sharing and Reusing Architectural Knowledge, 2009. SHARK'09. ICSE Workshop on*, 2009, pp. 57-64.

[11] S. Wagner, "Cost optimisation of analytical software quality assurance," 2007.

[12] C. Andersson and P. Runeson, "A replicated quantitative analysis of fault distributions in complex software systems," *Software Engineering, IEEE Transactions on,* vol. 33, pp. 273-286, 2007.

[13] T. Galinac Grbac, P. Runeson, and D. Huljenić, "A second replicated quantitative analysis of fault distributions in complex software systems," *Software Engineering, IEEE Transactions on,* vol. 39, pp. 462-476, 2013.

[14] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *Software Engineering, IEEE Transactions on,* vol. 26, pp. 797-814, 2000.

[15] K.-Y. Cai and B.-B. Yin, "Software execution processes as an evolving complex network," *Information Sciences,* vol. 179, pp. 1903-1928, 2009.

[16] S. Wagner, "Using economics as basis for modelling and evaluating software quality," in *Proceedings of the First International Workshop on The Economics of Software and Computation*, 2007, p. 2.

[17] F. P. Brooks and N. S. Bullet, "Essence and accidents of software engineering," *IEEE computer,* vol. 20, pp. 10-19, 1987.

[18] K. El Emam and A. G. Koru, "A replicated survey of IT software project failures," *Software, IEEE,* vol. 25, pp. 84-90, 2008.

[19] ISO/IEC/IEEE-15288, "ISO/IEC/IEEE 15288:2015(E) Systems and software engineering - System life cycle processes," ed: International Standards Organisation, IEC, IEEE, 2015.

[20] J. Bøegh, "A New Standard for Quality Requirements," *IEEE Software,* vol. 25, pp. 57-63, 2008.

[21] ISO/IEC-12207, "ISO/IEC 12207:2008(en) Systems and software engineering - Software life cycle processes," ed: International Standards Organisation, IEC, 2008.

[22] ISO-9001, "ISO 9001:2008(en) Quality management systems - Requirements," ed: ISO, 2008.

[23] C. P. Team, "CMMI for Development, Version 1.3 (CMU/SEI-2010-TR-033). Software Engineering Institute," ed: Carnegie Mellon University, 2010.

[24] A. Kovács and K. Szabados, "Test software quality issues and connections to international standards," *Acta Universitatis Sapientiae, Informatica,* vol. 5, pp. 77-102, 2013.

[25] G. Coleman and R. Verbruggen, *A quality software process for rapid application development*: Springer, 1998.

[26] J. Li, N. B. Moe, and T. Dybå, "Transition from a plan-driven process to scrum: a longitudinal case study on software quality," in *Proceedings of the 2010 ACM-IEEE international symposium on empirical software engineering and measurement*, 2010, p. 13.

[27] J. Sutherland, C. R. Jakobsen, and K. Johnson, "Scrum and cmmi level 5: The magic potion for code warriors," in *Hawaii International Conference on System Sciences, Proceedings of the 41st Annual*, 2008, pp. 466-466.

[28] M. Huo, J. Verner, L. Zhu, and M. A. Babar, "Software quality and agile methods," in *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, 2004, pp. 520-525.

[29] M. Fowler and J. Highsmith, "The agile manifesto," *Software Development,* vol. 9, pp. 28-35, 2001.

[30] A. Begel and N. Nagappan, "Usage and perceptions of agile software development in an industrial context: An exploratory study," in *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, 2007, pp. 255-264.

[31] N. E. Fenton and M. Neil, "Software metrics: roadmap," in *Proceedings of the Conference on the Future of Software Engineering*, 2000, pp. 357-370.

[32] T. M. Khoshgoftaar and N. Seliya, "Comparative assessment of software quality classification techniques: An empirical case study," *Empirical Software Engineering,* vol. 9, pp. 229-257, 2004.

[33] D. Gupta, H. K. Mittal, and V. Goyal, "Comparative Study of Soft Computing Techniques for Software Quality," *International Journal of Software Engineering Research and Practices,* vol. 1, pp. 33-37, 2011.

[34] J. Tian, "Quality-evaluation models and measurements," *Software, IEEE,* vol. 21, pp. 84-91, 2004.

[35] A. Wood, "Software reliability growth models," *Tandem Technical Report,* vol. 96, 1996.

[36] N. Bridge and C. Miller, "Orthogonal defect classification using defect data to improve software development," *Software Quality,* vol. 3, pp. 1-8, 1998.

[37] L. Huang, V. Ng, I. Persing, R. Geng, X. Bai, and J. Tian, "AutoODC: Automated generation of orthogonal defect classifications," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, 2011, pp. 412-415.

[38] A. L. Goel, "Software reliability models: Assumptions, limitations, and applicability," *Software Engineering, IEEE Transactions on,* pp. 1411-1423, 1985.

[39] R. E. Al-Qutaish, "Quality models in software engineering literature: an analytical and comparative study," *Journal of American Science,* vol. 6, pp. 166-175, 2010.

[40] S. Fahmy, N. Haslinda, W. Roslina, and Z. Fariha, "Evaluating the Quality of Software in e-Book Using the ISO 9126 Model,"

*International Journal of Control and Automation,* vol. 5, pp. 115-122, 2012.
[41] J. A. McCall, P. K. Richards, and G. F. Walters, "Factors in software quality. volume i. concepts and definitions of software quality," DTIC Document1977.
[42] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative evaluation of software quality," in *Proceedings of the 2nd international conference on Software engineering*, 1976, pp. 592-605.
[43] B. W. Boehm, J. R. Brown, and H. Kaspar, "Characteristics of software quality," 1978.
[44] R. B. Grady, *Practical software metrics for project management and process improvement*: Prentice-Hall, Inc., 1992.
[45] ISO/IEC-9126, "ISO/IEC 9126-1:2001 Software engineering -- Product quality -- Part1: Quality model," ed: ISO/IEC, 2001.
[46] G. R. Dromey, "A model for software product quality," *Software Engineering, IEEE Transactions on,* vol. 21, pp. 146-162, 1995.
[47] H.-W. Jung, S.-G. Kim, and C.-S. Chung, "Measuring software product quality: A survey of ISO/IEC 9126," *IEEE software,* pp. 88-92, 2004.
[48] ISO/IEC-25000, "ISO/IEC 25000:2014 Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - Guide to SQuaRE," ed: ISO, IEC, 2014.
[49] P. Parmakson, "Alignment of software quality and service quality," in *Advanced Information Systems Engineering*, 1995, pp. 355-365.
[50] I. Castillo, F. Losavio, A. Matteo, and J. Bøegh, "REquirements, Aspects and Software Quality: the REASQ model," *Journal of Object Technology,* vol. 9, pp. 69-91, 2010.
[51] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, 2007, pp. 30-39.
[52] D. Coleman, D. Ash, B. Lowther, and P. Oman, "Using metrics to evaluate software system maintainability," *Computer,* vol. 27, pp. 44-49, 1994.
[53] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*: Elsevier Science Inc., 1977.
[54] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, "Software fault prediction metrics: A systematic literature review," *Information and Software Technology,* vol. 55, pp. 1397-1418, 2013.

[55] V. R. Basili, "Software modeling and measurement: the Goal/Question/Metric paradigm," 1992.
[56] K. Mordal, N. Anquetil, J. Laval, A. Serebrenik, B. Vasilescu, and S. Ducasse, "Software quality metrics aggregation in industry," *Journal of Software: Evolution and Process,* vol. 25, pp. 1117-1135, 2013.
[57] B. Vasilescu, A. Serebrenik, and M. van den Brand, "By no means: A study on aggregating software metrics," in *Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics*, 2011, pp. 23-26.
[58] P. Oman and J. Hagemeister, "Construction and testing of polynomials predicting software maintainability," *Journal of Systems and Software,* vol. 24, pp. 251-266, 1994.
[59] A. Janes, M. Scotto, W. Pedrycz, B. Russo, M. Stefanovic, and G. Succi, "Identification of defect-prone classes in telecommunication software systems using design metrics," *Information sciences,* vol. 176, pp. 3711-3734, 2006.
[60] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," *Journal of Systems and Software,* vol. 83, pp. 2-17, 2010.
[61] M. R. Lyu, *Handbook of software reliability engineering* vol. 222: IEEE computer society press CA, 1996.
[62] V. Zeljković, N. Radovanović, and D. Ilić, "Software reliability: Models and parameters estimation," *Scientific Technical Review,* vol. 61, pp. 57-60, 2011.
[63] K. M. Faqih, "What is Hampering the Performance of Software Reliability Models? A literature review," in *International MultiConference of Engineers and Computer Scientists*, 2009.
[64] Y. S. Su, C.-Y. Huang, Y. S. Chen, and J. X. Chen, "An artificial neural-network-based approach to software reliability assessment," *TENCON 2005 2005 IEEE Region 10,* pp. 1-6, 2005.
[65] A. M. Neufelder, *Ensuring software reliability*: CRC Press, 1992.
[66] S. S. Gokhale, P. N. Marinos, and K. S. Trivedi, "Important milestones in software reliability modeling," in *SEKE*, 1996, pp. 345-352.
[67] L. Shanmugam and L. Florence, "A comparison of parameter best estimation method for software reliability models," *International Journal of Software Engineering & Applications,* vol. 3, pp. 91-102, 2012.

[68] C. Zheng, X. Liu, S. Huang, and Y. Yao, "A Parameter Estimation Method for Software Reliability Models," *Procedia Engineering,* vol. 15, pp. 3477-3481, 2011.

[69] C.-W. Chang, C.-R. Wu, and H.-L. Lin, "Integrating fuzzy theory and hierarchy concepts to evaluate software quality," *Software Quality Journal,* vol. 16, pp. 263-276, 2008.

[70] A. Stefani, M. Xenos, and D. Stavrinoudis, "Modelling e-commerce systems' quality with belief networks," in *Virtual Environments, Human-Computer Interfaces and Measurement Systems, 2003. VECIMS'03. 2003 IEEE International Symposium on*, 2003, pp. 13-18.

[71] K. Kaswan, S. Choudhary, and K. Sharma, "Software Reliability Modeling using Soft Computing Techniques: Critical Review," *J Inform Tech Softw Eng,* vol. 5, p. 2, 2015.

[72] H. Yang, "Measuring Software Product Quality with ISO Standards Base on Fuzzy Logic Technique," *Affective Computing and Intelligent Interaction,* pp. 59-67, 2012.

[73] S. Wagner and F. Deissenboeck, "An integrated approach to quality modelling," in *Proceedings of the 5th International Workshop on Software Quality*, 2007, p. 1.

[74] S. Wagner, "A literature survey of the quality economics of defect-detection techniques," in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, 2006, pp. 194-203.

[75] S. Wagner, "A Bayesian network approach to assess and predict software quality using activity-based quality models," *Information and Software Technology,* vol. 52, pp. 1230-1241, 2010.

[76] N. Fenton and M. Neil, "Software metrics and risk," in *Proc 2nd European Software Measurement Conference (FESMA'99), TI-KVIV, Amsterdam, ISBN*, 1999, pp. 90-76019.

[77] N. Fenton, M. Neil, W. Marsh, P. Hearty, D. Marquez, P. Krause*, et al.*, "Predicting software defects in varying development lifecycles using Bayesian nets," *Information and Software Technology,* vol. 49, pp. 32-43, 2007.

[78] D. Marquez, M. Neil, and N. Fenton, "Improved reliability modeling using Bayesian networks and dynamic discretization," *Reliability Engineering & System Safety,* vol. 95, pp. 412-425, 2010.

[79] C. Van Koten and A. Gray, "An application of Bayesian network for predicting object-oriented software maintainability," *Information and Software Technology,* vol. 48, pp. 59-67, 2006.

[80] E. Kornyshova, R. Deneckère, and C. Salinesi, "Using Multicriteria Decision-Making to Take into Account the Situation in System Engineering," in *CAISE'08 forum*, 2008, p. 25.
[81] M. Grabisch, "The application of fuzzy integrals in multicriteria decision making," *European journal of operational research,* vol. 89, pp. 445-456, 1996.
[82] J. Bragge, P. Korhonen, H. Wallenius, and J. Wallenius, "Bibliometric analysis of multiple criteria decision making/multiattribute utility theory," in *Multiple criteria decision making for sustainable energy and transportation systems*, ed: Springer, 2010, pp. 259-268.
[83] A. Barcus and G. Montibeller, "Supporting the allocation of software development work in distributed teams with multi-criteria decision analysis," *Omega,* vol. 36, pp. 464-475, 2008.
[84] I. Stamelos and A. Tsoukias, "Software evaluation problem situations," *European Journal of Operational Research,* vol. 145, pp. 273-286, 2003.