ICEBERG

How to estimate costs of poor quality in a Software QA project: a novel approach to support management decisions

Industry-Academia Partnerships and Pathways (IAPP)
Call: FP7-PEOPLE-2012-IAPP

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n°324356

Deliverable No.:	**2.1**
Deliverable Title:	Industrial needs collection & state of the art surveys
Organisation name of lead Contractor for this Deliverable:	CINI
Author(s):	Roberto Pietrantuono Stefano Russo Ida Licia Battipaglia Claudio Gaiani Luis Fernandez Daniel Rodriguez
Work package contributing to the deliverable:	2
Task contributing to the deliverable:	T 2.1, T2.2, T2.3
Total N. of Pages	64

Deliverable D2.1: "Industrial needs collection & state of the art surveys"

Table of Versions

Version	Date	Version Description	Contributors
0.1	11-07-2013	Document Structure. Background, Content on "Background", "Framework", and "Quality Standard" Sections.	Roberto Pietrantuono
0.2	06-09-2013	Updated Structure. Content on "Quality of the product" section.	Roberto Pietrantuono Stefano Russo
0.3	17-10-2013	Contributions by other partners integrated ("Factors impacting quality and cost" and "quality based decision making process" sections).	Roberto Pietrantuono, Luis Fernandez, Daniel Rodriguez, Claudio Gaiani Ida Licia Battipaglia
1.0	30-10-2013	Formatting refinements. Document completed	Roberto Pietrantuono

TABLE OF CONTENTS

1 EXECUTIVE SUMMARY

The ICEBERG project intends to carry out an intensive Transfer of Knowledge (ToK) in the Software Quality Assurance domain. To this aim, the project pursues the following high level objectives: *i)* investigation of what is the current state of the art about software quality measurement and analysis, software quality assurance, software quality standards, software quality improvement means and strategies; *ii)* definition of solutions for:

- identifying critical activities from the cost/quality point of view;
- estimating quality and related cost associated with the implementation of quality assurance activities and with the missing, incomplete or wrong implementation of such activities;
- monitoring and controlling quality by data collection and analysis; implementing support means on decision-making on the next steps at quality management level.

This document is the deliverable D2.1 of the ICEBERG project – "Industrial needs collection and state of the art surveys". The objective of this document is to provide practitioners, working in the quality assurance and software process improvement area, with an overview of existing works concerning the main areas of software quality, as well as to collect the actual industrial needs in the field of quality assurance. The document will provide the ICEBERG partner with the guidelines to pursue the mentioned project goals.

2 OVERVIEW

This document explores the state of the state of the art and of the practice in the software quality area, both from the literature point of view and from the industrial perspective. After a background section, meant to establish a common vocabulary for projects' partners and for document readers, an introductory section follows, which describes a generic high-level framework for quality/cost measurement and analysis helpful to contextualize the successive sections and future project's activities as well. The subsequent sections will then focus on:

1. **Quality standards**: What are the most common software process/product quality standards and models? How do they define quality, and how propose to pursue quality objectives?
2. **Attributes and Metrics:** What are the main attributes of quality of software products and software process, and the most used metrics to measure such attributes?
3. **Quality-related Factors**: What are the main external factors that impact the quality of the developed product?
4. **Decision-making process:** How is the current decision-making process supported in industry, and what are their needs with respect to this problem?

The review is oriented to provide feedback, besides to interested practitioners, to the subsequent phases of the project, in which proper standards, metrics, analysis strategies, and industry requirements will be used for defining the quality model and the decision-making support method.

3 BACKGROUND

3.1 TERMINOLOGY

In the wide field of software quality, there are several different uses of terms and definitions, sometimes in contrast to each other especially if used in different contexts (e.g., academia, industry). The aim of this Section is to set a common use of terms throughout the project.

Software Quality: Software Quality is (1) the degree to which a system, component, or process meets specified requirements. (2) The degree to which a system, component, or process meets customer or user needs or expectations (IEEE 610.12).

Software quality assurance, software quality control, and software quality engineering are a planned and systematic set of activities to ensure quality is built into the software.

Software Quality Assurance (SQA): The function of software quality that assures that the standards, processes, and procedures are appropriate for the project and are correctly implemented (NASA , 2009).

Software Quality Control: The function of software quality that checks that the project follows its standards, processes, and procedures, and that the project produces the required internal and external (deliverable) products (NASA , 2009).

Software Quality Engineering: The function of software quality that assures that quality is built into the software by performing analyses, trade studies, and investigations on the requirements, design, code and verification processes and results to assure that reliability, maintainability, and other quality factors are met (NASA , 2009).

Quality attribute. A feature or characteristic that affects an item's quality. Syn: quality factor. (IEEE 610.12). Note: ISO 9126 (ISO/IEC 9126, 2001) distinguishes between (sub-)characteristic and attribute, the latter being at lower level and a "measurable" (sub-)characteristic. We do not apply this distinction here.

Measure. A quantitative indication of the extent, amount, dimensions, capacity or size of some attribute of a product or process. (IEEE 729-1993, 1993)

Metric. A quantitative measure of the degree to which a system, component, or process possesses a given attribute (IEEE 729-1993, 1993) *and* (IEEE 610.12).

Measurement. The process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules. (Fenton & Pfleeger, 1998)

In the following, definitions about fault, defect, error, and failure are reported: since there are two fundamental ways of defining these, both are reported in order to be aware of their use throughout the project and avoid each one coming up with his own definition, creating ambiguities.

According to IEEE standards (software engineering community):

Error: An error is a mistake, misconception, or misunderstanding on the part of a software developer.

Defect: An imperfection or deficiency in a work product where that work product does not meet its requirements or specifications and needs to be either repaired or replaced (IEEE 1044-2009, 2009). According to this standard, a Fault is a manifestation of an error in software. A fault is a subtype of the supertype defect. A defect is a fault only if it is encountered during software execution (thus causing a failure);

Fault: A fault (defect) is introduced into the software as the result of an error. It is an anomaly in the software that may cause it to behave incorrectly, and not according to its specification. *In this case fault and defect are synonymous.*

Failure: A failure is the inability of a software system or component to perform its required functions within specified performance requirements.

According to the community of "dependability" and "fault tolerance":

Fault. A fault is the adjudged or hypothesized cause of an error. A fault is active when it produces an error, otherwise it is dormant. (Avizienis, Laprie, Randell, & Landwehr, 2004)

Error. An error is that part of the system state that may cause a subsequent failure: a failure occurs when an error reaches the service interface and alters the service. (Avizienis, Laprie, Randell, & Landwehr, 2004)

Failure. A system failure is an event that occurs when the delivered service deviates from correct service. (Avizienis, Laprie, Randell, & Landwehr, 2004)

There is a substantial agreement on Failure and on Fault term (which corresponds roughly to defect): *the fundamental difference is that an error for IEEE definitions comes before the defect (fault), because it is the human mistake; for dependability definitions, an error is a state consequent to a fault activation* (it captures the state propagation from the fault activation to the failure manifestation). In the former case the chain is: **Error -> Defect -> Failure, in the second case is: Fault -> Error -> Failure.**

Bug: a bug is software defect (or fault) introduced during coding.

Issue: an issue is a notification of a deviation of the provided service from the expected one as perceived by the end user (i.e., from what user believes to be the correct service). In other words, it may also not correspond to an actual defect.

Verification: Confirmation by examination and provision of objective evidence that specified requirements have been fulfilled (ISO/IEC 12207-2008, 2008). In

other words, verification ensures that "you built it right". It applies to any intermediate artefact, since it check conformance between an artefact and its specification (e.g., a low-level design against a low-level design specification).

Validation: Confirmation by examination and provision of objective evidence that the particular requirements for a specific intended use are fulfilled (ISO/IEC 12207-2008, 2008). In other words, validation ensures that "you built the right thing". Validation is against "user requirements", verification is against "specified requirements".

Independent Verification and Validation (IV&V): *Verification and validation performed by an organization that is technically, managerially, and financially independent.*

Testing: process of executing the software to reveal malfunctioning (i.e., to expose failures). Testing is not to find bugs, but to expose failures. The activities aimed at finding bugs is known as bug localization; the activities aimed at removing bugs (or at finding and removing bugs) is known as debug.

Analysis (Technique): set of activities aimed verifying a property of the software. (Pezze & Young, 2008)

Static Analysis: Analysis activity acting on static artefacts, for instance on the source code. (Pezze & Young, 2008)

Dynamic Analysis: Analysis activity acting on dynamic artefacts, namely one the software execution traces. (Pezze & Young, 2008)

Test Case: A test case is a set of inputs, execution conditions, and a pass/fail criterion. (Pezze & Young, 2008) *(This usage follows the IEEE standard.)*

Test case specification: A test case specification is a requirement to be satisfied by one or more actual test cases. (Pezze & Young, 2008) *(This usage follows the IEEE standard.)*

Test suite: A test suite is a set of test cases. Typically, a method for functional testing is concerned with creating a test suite. A test suite for a program, system, or individual unit may be made up of several test suites for individual modules, subsystems, or features. (Pezze & Young, 2008) *(This usage follows the IEEE standard.)*

Test or test execution: The activity of executing test cases and evaluating their results. When we refer to "a test," we mean execution of a single test case, except where context makes it clear that the reference is to execution of a whole test suite. (Pezze & Young, 2008) *(The IEEE standard allows this and other definitions.)*

3.2 MAIN QUALITY ATTRIBUTES AND IMPACTING FACTORS

The main drivers of the ICEBERG project are the **quality** of the software product and **cost**. In particular, we aim at understanding:

- What are the **factors** into an organization (e.g., a company) that mostly **impact the (poor) quality** of the final software product. This implies investigating many aspects of the process, of the organization, of the environment, and in general of the *context* in which the product is built.
- What is the relationship between such quality-impacting factors and cost factors, and how this can be 1) measured, 2) controlled, 3) improved. We distinguish between two types of quality-cost relation:
  1. a **direct relation from quality factors to cost to achieve that quality** (e.g., measuring effort and time to develop the product with that quality);
  2. an **indirect relation between "missed" quality and cost,** i.e., what is the cost associated with poor quality, or conversely how much the bad quality costs (e.g., in terms of user dissatisfaction, maintenance costs due to residual defects, etc.).

To this aim, we present, in the following, our high-level view of software quality and the basic framework to measure, control, improve quality/cost, which will be refined, customized, and tailored for specific contexts, alongside the project.

Quality attributes of a software product (i.e., **product quality**) can be:

Internal: "static" properties of a software artefact (requirements specification, design specification, code) as seen by the *developers/managers* within the organization. They are static because their measurement does not need the execution of the software in its environment.

Examples are: *Requirements size, volatility, completeness, design decoupling, modularity, degree of reuse, code McCabe complexity, size in lines of code, fan-in, fan-out.*

External: properties of the software product that the *final users* can experience and enjoy. It expresses therefore the dynamic behaviour of the software in its usage context.

Examples are: *Reliability, functional correctness, robustness, usability, portability, interoperability.*

Note: ISO 9126 (ISO/IEC 9126, 2001) *also introduces the "in-use" attributes, referring to efficacy, efficiency, safety, and satisfaction with which the software meet the user needs, and is related to the user perception. We neglect this distinction between external and in-use quality the latter is represented by too abstract and implicit attributes that are generally desirable for any software*

system, and are therefore useless in the practice (rather, engineers activities are aimed at pursuing one or more of the "external" quality attributes).

One or more external attributes represent the desired characteristics of the final software product, and the organization and the overall process are oriented to provide high level of such quality attributes at low cost.

External quality attributes are particularly difficult to measure. Thus, among the many attributes, and consequent metrics, of potential interest, a special role is played by the ***degree of defectiveness***. In fact, we consider defectiveness, at any level, as generic measure of (non-)quality of the artefact and/or of a process phase/activity, since it is the closest measurable attribute to user-perceived external quality, and the base for measuring more complex external attributes (e.g., reliability, availability, usability, etc.). Defectiveness can be measured by a set of metrics, the most relevant one being the number of defects (but also the defect density, the estimated final number of defects). Moreover, it can be characterized a set of sub-attributes, useful for analysis purposes, such as the type of defects, the phase injection origin, the trigger, the detection phase, the source of the defects, and so on.

Internal quality impacts defectiveness and thus external quality. The measurements of internal quality is therefore useful not only for understanding the quality of intermediate product, but also for external quality prediction (e.g., in terms of expected defects) and the consequent planning of (design and testing) activities.

But both internal and external quality, as the final quality/cost relationships, are heavily impacted by ***non-product quality factors, namely by factors related to the organization and the context in which the product is developed*** *(as depicted in Figure 1).*

So, we have the following high-level view as depicted in Figure 1, showing some (not all) of the factors potentially impacting the product quality.

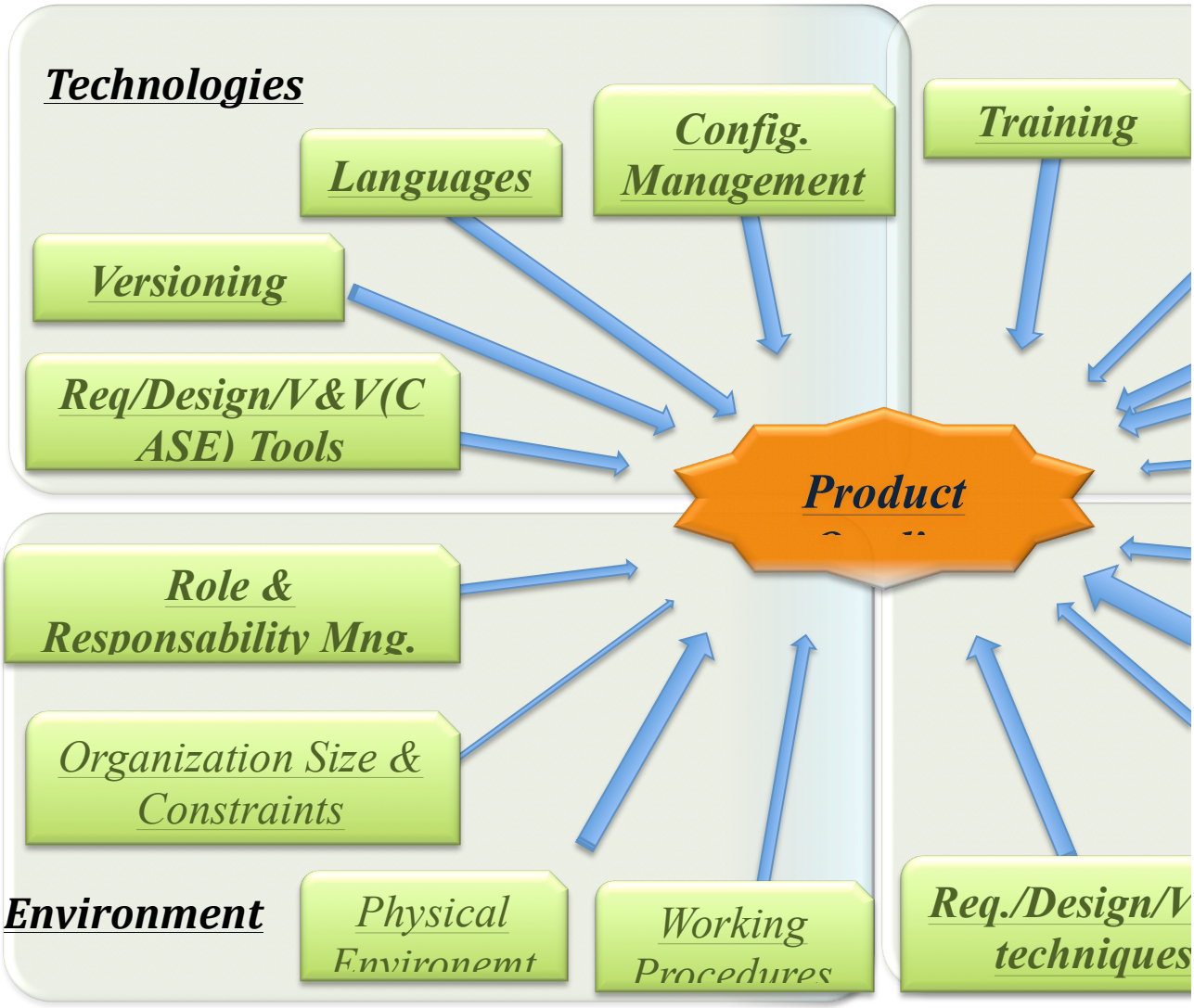

Figure 1. Some of the factors impacting the product quality

All these and others factors describing the context in which the product is created, may have a great impact on product quality.

Thus, summarizing, we distinguish:

1. **Internal quality** attributes**;**
2. **External quality** attributes**;**
    a. **Defectiveness** as reference attribute for product quality measurements**;**
3. **Context-related quality** attributes.

Impacts on product quality translates into impact on the final cost; however, the relationship between quality and cost, or to better say between non-quality and cost, need to be assessed both by reviewing literature studies, and by customized field data (since literature data embody a too wide spectrum).

4 A GENERIC FRAMEWORK FOR MEASUREMENT AND ANALYSIS

A general scheme to capture relationships among quality factors is reported in Figure 2.

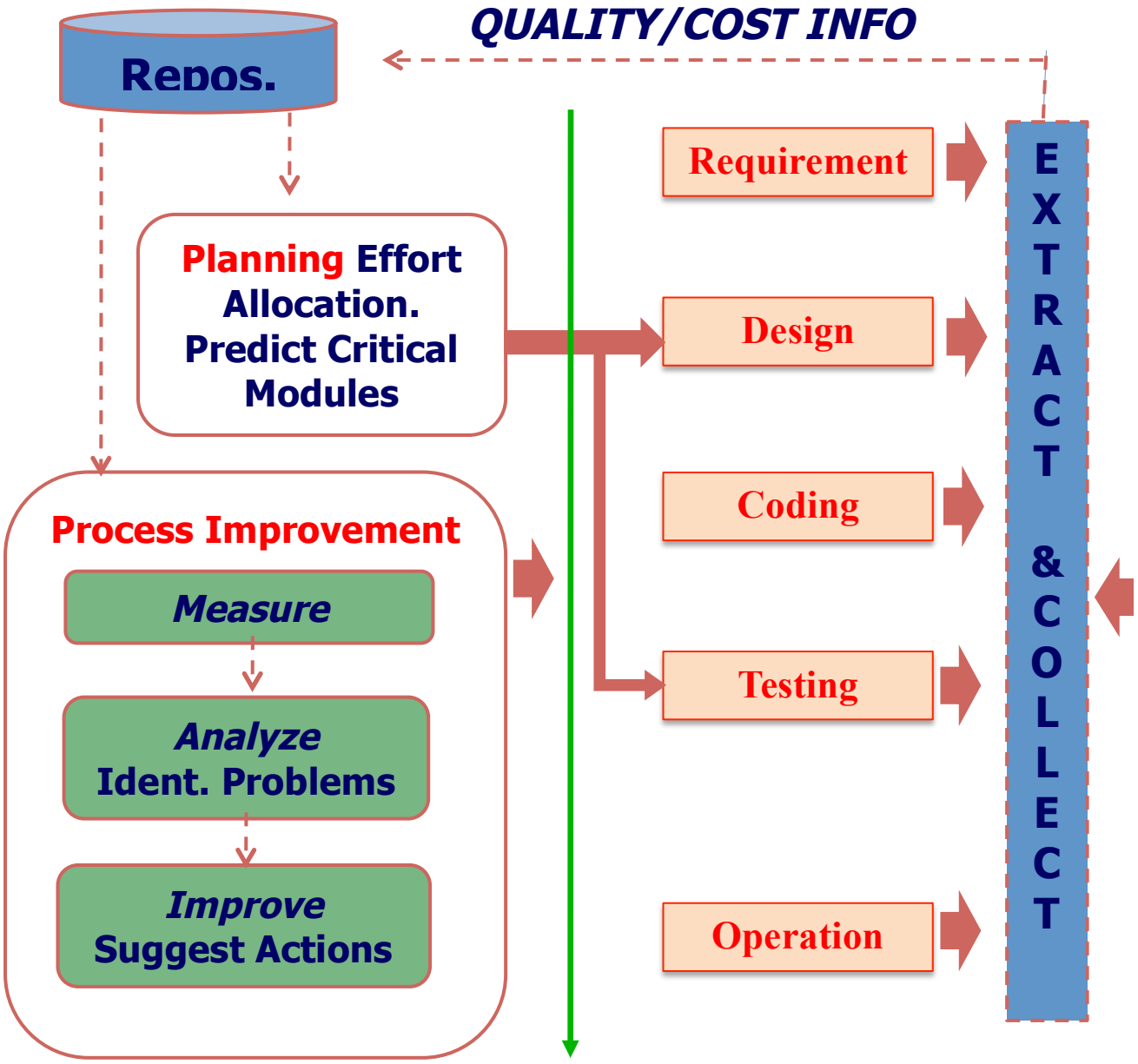

Figure 2: The general scheme for measurement and analysis

In this scheme, the sources of information considered are:

- Artefacts (documents, models, code) at each phase/activity of the process development of each available product in the company
- Context-related information, as the ones in Figure 1.

From the activities carried out in each phase it is possible to extract and collect relevant information for the subsequent analysis and improvement steps. It is important to remark that the amount of information that can be extracted is potentially very large: thus, the first thing that the analyst should do is to figure out what is really relevant in relation to his objective. Several paradigm can be adopted, such as the *goal-question-metric*, or the *DMAIC method from six-sigma*, in order to define proper metrics in relation to objectives. The general steps are in any case:

- Define the objective of the measurement;
- Select the (sub-)processes to monitor, analyze improve to satisfy the objective;
- Select the attributes (among the many existing ones) and the corresponding metrics, trying to obtain the smallest useful subset.

In the scheme, metrics collected from single phases capture *static characteristics* of any intermediate product artefact (e.g., requirements, design, code), namely internal quality attributes; whereas metrics collected from the right side represent *characteristics of the context/organization that are independent from development phases.*

Internal quality characteristics are often supposed to adhere to some reference values, denoting a "good" acceptable quality of the intermediate artefact.

Both these and context-related attributes are also typically supposed to impact the final defectiveness and quality/cost trade-off. More advanced approaches use (some of) these static metrics to build prediction models, in order to estimate the defectiveness of software modules. This activity may be a good driver to properly concentrate efforts on modules predicted to be the most defective ones, and thus to plan design and especially V&V activities.

Thus, all the collected information in the loop may be used basically at two different and complementary levels:

- **Product-level;** in this case metrics are used to measure the quality of the artefacts of the single product (both in terms of internal quality or as number of defects). When complemented with information regarding the "size" of what produced and cost metrics such as the "effort" to produce it (e.g., by people/month, schedule), measures of productivity can be obtained besides quality. Both static and context-related metrics can be used for prediction purposes (e.g., predict defectiveness), and thus as support for planning and scheduling activities driven by expected defectiveness.
- **Process/Management-level;** in this case, there are two ways of using the information to measure the quality of the development process, and of the organization at higher level, and its relation with cost:
  - **Trend analysis** of (normalized) quality and cost attributes over products, along with their variability across products. This is the main way to evaluate the quality of the process through its products.

- **Effectiveness per process phase/activity:** this is a special case of analysis, in which defects data are considered per phase/activity, in order to judge, across product, what activities are more critical from the quality point of view.

- **Impact analysis** of single (internal and context-related) attribute on quality and on cost. This includes both internal quality attributes, such as lines of code, complexity, size of requirements, decoupling, and so on, and phase-independent organizational metrics (e.g., human factor metrics), useful especially for cost analysis. This evaluation needs typically to be done over more products, and determine process-, management-level choices. The purpose is to infer relations and/or perform predictions.
  Note that both types quality/cost relationship have to be evaluated: 1) the **direct relation from quality attributes to cost** attributes to achieve that quality (e.g., measuring effort and time to develop the product with that quality), and 2) the **indirect relation between "missed" quality and cost**: this can be obtained be a two-level approach, which first relates internal/context quality to quasi-external (e.g., defectiveness) quality, and then relates, in a second step, the "missed" quality (in terms of residual defectiveness) to the cost that it causes (e.g., by the cost of maintenance intervention).

This general high-level scheme is meant to provide practitioner with an overall picture of measurement and analysis activities for quality and cost management. The view will serve also the ICEBERG partners as a general frame and possible guideline to conduct their activity within the project scope. In the following

5 QUALITY STANDARDS

Software quality assurance is by now one of the most important and expensive activities within a software development lifecycle. Suffice it to recall that the V&V phase, which is a crucial activity in the quality assurance framework, may account, in terms of both time and resources used, for more than 50% of the cost of the entire development. Thus, issues related to the definition and implementation of appropriate quality assurance strategies go far beyond the purely technical and / or technology aspect; they involve, in a significant way, methodological aspects, typically related to the production process, to the development process, as well as to organizational structure and business strategies for medium and long term.

In the ICEBERG project, software quality standards play a key role for the foreseen process definition. In this Section, the most important product/process quality standards (such as ISO/IEC 15504, ISO/IEC 29119, ISO 9126) and models (such as the CMMI) will be reviewed, in order to identify those elements of interest to the project purposes. Guidelines suggested will be considered for outlining the framework within which the models-based process will be defined.

Process-oriented Standards

Due to the increasing importance of quality, several methodologies and standards have been defined aiming at defining quality attributes and suggesting organizational frameworks to measure and improve those attributes. In the following the most commonly used methodologies and standards are briefly surveyed. World-wide organizations working on software proposed methods for quality control and evaluation of activities, and for the test process as well. We intend to integrate their most important concepts, such as the ability of "measuring" the process, of monitoring its state, and of exploiting the feedback obtained for improving the process in the development of future products.

Some standards rule about the entire software development process, including the V&V phase. One of the most relevant is the ISO/IEC 15504 (ISO/IEC 15504, 2012), also known as SPICE (Software Process Improvement and Capability Determination). It acts as reference model for the so-called “maturity models”, in which an assessor aims at giving a judgment of the organization’s capabilities for delivering products. ISO/IEC 15504 is the reference model for the maturity models (consisting of capability levels which in turn consist of the process attributes and further consist of generic practices) against which the assessors can place the evidence that they collect during their assessment, so that the assessors can give an overall determination of the organization's capabilities for delivering products.

The reference model defines a process dimension and a capability dimension; means for verifying the adherence to such a model are defined in the standard. The process dimension refers to an external process lifecycle standard, named ISO/IEC 12207 (ISO/IEC 12207-2008, 2008), which defines the core processes involved in the development of a software product: “acquisition” in the initial phase, “supply”, in which a project management plan is developed, “development”, including requirements definition, high-level and module design,

coding, module, integration and system test, “operation”, and “maintenance”. Processes are divided in five categories: “customer/supplier”, “engineering”, “supporting”, “management”, “organization”, according to the type of process to be implemented. The capability dimension defines levels of process maturity. The following levels are defined: “Optimizing process”, “Predictable process”, “Established process”, “Managed process”, “Performed process”, “Incomplete process” (the former being the best level) measured by several process attributes.

ISO/IEC 15504 can be used to perform process improvement within a technology organization. It has been successful so far, and other standards are derived from it (e.g., the Automotive SPICE, for the automotive domain) and is reference for many other techniques and models.

A very successful maturity model is the Capability Maturity Model Integrated (CMMI) (Software Engineering Institute, 2010). CMMI, and its appraisal standard SCAMPI (Standard CMMI Appraisal Method for Process Improvement), support the improvement of the development process by indicating a set of best practices for each development and maintenance activity. It defines 5 levels, which inspired also ISO 15504, indicating the maturity of the process, and a set of "process areas" to implement according to the level of maturity (also for these attributes, levels of "capability" are foreseen).

Starting from this idea, a similar model has been conceived specifically for the testing phase: the Testing Maturity Model has been defined for supporting the entire V&V process. It grounds on the same concepts of the CMMI, detailing them for V&V. It also includes 5 levels of maturity, with the following objectives:

Level 1, Initial: no specific objective; Level 2, Managed: the testing process is managed and clearly separated from debugging; there is a testing strategy; the process is controlled and is focused on functional testing; Level 3, Defined: the testing process is no longer considered as a step after the implementation, but it is integrated in the development process; the testing is also non-functional; Level 4, Management and Measurement: it is developed a process for quality evaluation of the software, and a measurement program of the testing; it is also developed a program of revision of the entire organization; Level 5, Optimization: the testing process is optimized; there are implemented measures of quality; it is applied the defect prediction based on the available data, and on the feedback received from the process application to a product.

A fundamental feature for optimized levels of these standards is the ability of monitoring and measuring the process in order to propose improvements: companies are required to learn from the past, in order to continuously improve future products.

Another standard concerning quality is the IEEE 1012-2012 (IEEE 1012, 2012), a standard for software Verification and Validation recently updated. It is a process standard that addresses all system and software life cycle processes including the Agreement, Organizational Project-Enabling, Project, Technical, Software Implementation, Software Support, and Software Reuse process groups. This standard defines the verification and validation processes that are

applied to the system, software, and hardware development throughout the life cycle, including acquisition, supply, development, operations, maintenance, and retirement. This standard applies to the system, software, and hardware being acquired, developed, maintained, or reused.

This standard is organized into clauses (Clause 1 through Clause 12), tables, figures, and annexes. Clause 2 through Clause 12 provide (with some Tables) the mandatory V&V requirements for this standard. Each clause containing V&V activities and tasks has a subset of tables associated with the V&V requirements for that clause.

Clause 1 provides guidance for using this standard. Clause 2 is reserved for normative references; however, this standard does not prescribe any normative references. Clause 3 provides a definition of terms, abbreviations, and conventions. Clause 4 describes the relationships between the V&V processes and the life cycle processes from ISO/IEC 15288:2008 and ISO/IEC 12207:2008, and it describes how the V&V standard is applied recursively within the concept of a system of systems and from system to software or hardware components. Clause 5 describes the use of integrity levels to determine the scope and rigor of V&V processes. Clause 6 explains how V&V tasks are described within this standard. Clause 7 describes common V&V tasks; that is, tasks that are common across application of this standard to system, software, or hardware. Clause 8 describes system V&V tasks. Clause 9 describes software V&V tasks. Clause 10 describes hardware V&V tasks. Clause 11 describes V&V reporting, administrative, and documentation requirements. Clause 12 describes the content of a V&V plan.

Another new international standard by ISO/IEC specifically on software testing is being defined, which aims to replace old partial or incomplete documents on single pieces of the V&V process with a definitive standard: it is the ISO/IEC 29119 (ISO/IEC 29119, 2013). The standard is defining several sections about: vocabulary, test process, test documentation, test techniques, and a process assessment model for software testing that can be used within any software development life cycle.

Our project is in line with the process organization that will be defined by this standard, in that it foresees to include aspects related to planning, organization, and management. Standard's levels cover these aspects: the organisational test strategies and policies, the management of testing projects including the design of project/level test strategies and plans and monitoring and controlling testing, and a dynamic test process that provides guidance for test analysis and design, test environment set-up and maintenance, test execution and reporting.

Part 2 of the standard defines a generic testing process model that can be used within any software development and testing life cycle. This process will be based on a four-layer testing process covering: Organizational Test Specifications (e.g. Organizational Test Policy, Organizational Test Strategy), Test Management (e.g. project test management, phase test management), Dynamic Test Processes, including test design & implementation, test environment set-up & maintenance, test execution and incident reporting.

Finally, it is worth to point out that guidelines for quality assurance are also proposed by domain-specific standards for software certification, such as the DO178B/C (RTCA - DO 178C, 2011) for the avionic domain, the CENELEC 50126 (CENELC 50126, 1999), 50128 (CENELEC 50128, 2011), 50129 (CENELEC 50129, 2003) for the railway domain, the ISO 26262 (ISO 26262, 2011) for the automotive, and so on. These usually prescribe process activities, and documentation to be produced, at a high level of abstraction. They are quite generic guidelines, as also in the previous cases, and the level of details by which they are implemented in the company is the result of a trade-off with the company needs in terms of cost and time. As for any other standard, we will consider the valuable guidelines provided by them; however, it is a cost-effective implementation of these guidelines that stimulates companies to conceive new solutions for instantiating their quality process, according to their need, to their organization (size, personnel skill, assets, know how), and to the features of their products. The ICEBERG objective is therefore to propose effective solutions to setup such an implementation. Thus, the project intends to move within the framework defined by the mentioned standards, but at the same time it has to implement effective solutions for make the standard guidelines concrete actions in a real process.

For instance, for quality monitoring/measurements and feedback/control of the process through decision-making support, we will define a specific strategy based on selected metrics and predictive models able to drive assurance activities and the "measurable" provided quality.

Product-oriented Standards

Besides standards guiding through the process, another relevant aspect to consider is about the product. In fact, when we talk about quality assurance, and improvement of process to pursue quality, it should be well intended what "quality" is meant for in the considered context.

It is thus important to figure out what attributes of quality will be relevant for a given context. One of the most important reference models for defining "quality" is the ISO/IEC 9126 standard (ISO/IEC 9126, 2001).

ISO 9126 is an international standard for the evaluation of software. It is divided into four parts, which addresses, respectively, the following subjects: quality model; external metrics; internal metrics; and quality in use metrics. ISO 9126 Part one, referred to as ISO 9126-1 defines a set of software quality characteristics.

ISO9126-1 aims at characterizing software for the purposes of software quality control, software quality assurance and software process improvement. The ISO 9126-1 software quality model identifies 6 main quality characteristics: *Functionality, Reliability, Usability, Efficiency, Maintainability, Portability*. It also defines 27 sub-features measurable via metrics. The standard defines a four-level model, defining the quality by three perspectives (external, internal, and in use) that each project should satisfy, the attributes qualifying the product according to the three defined perspective, for each attribute, the sub-features

(measureable requirements) representing it, and the metrics to perform measurements, which however are not always clear and unambiguous.

Product and process metrics are also defined in the literature to measure product's features describing the quality or features potentially related to the quality; this will be the subject of the next section.

6 QUALITY OF THE PRODUCT

This section aims at surveying the metrics available in the literature for measuring both software product and process quality. The main attributes regarding the internal and external quality of a product (e.g., reliability, functionality, usability, robustness) will be examined. Besides the usefulness for practitioners, this survey will help the project's partners to figure out which metrics should be considered for implementing an efficient quality/cost process.

6.1 INTERNAL QUALITY ATTRIBUTES AND METRICS

As defined by the described standards, software quality can be measured from an external, user-perceived, perspective, and from internal perspective. Clearly, the ultimate goal of any engineering activity is to provide a product with the desirable quality attributes as viewed by the customer, namely the external quality; but such a quality is heavily affected by the quality of any previous artefact within the development process. In fact, internal quality attribute just measure how good the product is at a previous stage before the final one: the better it is, the better the final product will be.

Thus, in the practice, many software engineers, mangers, and practitioners, rely on internal quality measures to control and thus drive the development activities, in order to end up with the desired external quality with the planned cost.

This sub-section presents those works that defined and/or used some of the most common internal quality attributes to quality measurement and process improvement.

The most studied set of internal metrics are the ones referring to the source code. Relatively fewer studies investigated requirements and design metrics. Code Metrics have been widely used in the past both to measure the quality of the code according to some reference values, and as predictors of software module defectiveness. The most commonly adopted metrics can be divided in: method-level, class-level, file-level, component-level metrics (Catal & Diri, 2009). Examples of most common method-level metrics are *McCabe* metrics (e.g., McCabe LoC, McCabe cyclomatic complexity) (McCabe, 1976), and *Halstead's* (software science) metrics (e.g., Volmue, Length, Difficulty) (Halstead, 1977). Class-level metrics are more recent, and refer to object oriented paradigm. Examples are the Chidamber–Kemerer (CK) metrics suite (Chidamber & Kemerer, 1994) proposed in 1994, the MOOD (metrics for object-oriented design) (Abreu & Carapuca, 1994), QMOOD (quality metrics for object-oriented design) (Bansiya & Davis, 2002), and Lorenz and Kidd (L&K) metric suites (Lorenz & Kidd, 1994). However, CK metrics suite is much more popular than other suites and they are mostly used if class-level metrics are applied. They are: *Coupling_Between_Objects (CBO), Depth_Of_Inheritance_Tree (DIT), Lack_Of_Cohesion_Of_Methods (LCOM), Num_Of_Children (NOC), Response_ For_Class (RFC), and Weighted_Method_Per_Class (WMC)*. WMC is the sum of the

complexities of all class methods. DIT is the distance of the longest path from a class to the root in the inheritance tree. RFC is the number of methods that can be executed to respond a message. NOC is the number of classes that are direct descendants for each class. CBO is the number of non-inheritance-related classes to which a class is coupled. LCOM is related to the access ratio of attributes. According to several software fault prediction studies (Zhou & Leung, 2006), CBO, WMC, and RFC are the most significant CK metrics for fault prediction.

Metrics per source file are also increasingly used for fault prediction, such as in (Khoshgoftaar, Gao, & Szabo, 2001), (Ostrand, Weyuker, & Bell, 2005): some of these metrics are the number of lines of code per file, number of lines of commented code per file, number of changes per file.

Much literature has been produced to find a "best" set of metrics able to predict defect proneness of software modules.
Much work is on investigating relationships between several kinds of software metrics and the defect proneness in a program. Early research was focused on the definition of metrics able to measure the complexity of a software module and, in turn, its likelihood to be faulty (such as the McCabe's and Halstead's metrics). Fault prediction approaches have then evolved by adopting machine learning and data mining algorithms and techniques, in order to establish a more accurate relationship between sets of software metrics and faults, using classifiers and regression models.
In (Gokhale & Lyu, Regression Tree Modeling for the Prediction of Software Quality, 1997), authors used a set of 11 metrics and an approach based on regression trees to predict faulty modules. In (Nagappan, Ball, & Zeller, 2006), authors mine metrics to predict the amount of post-release faults in five large Microsoft's software projects. They adopted the well-known statistical technique of Principal Component Analysis (PCA) in order to transform the original set of metrics into a set of uncorrelated variables, with the goal of avoiding the problem of redundant features (multicollinearity). The study in (Denaro, Morasca, & Pezze, Deriving Models of Software Fault-proneness, 2002), then extended in (Denaro & Pezze, An Empirical Evaluation of Fault-proneness Models, 2002) adopted logistic regression to relate software measures and fault-proneness for classes of homogeneous software products. Subsequent studies confirmed the feasibility and effectiveness of fault prediction using public-domain datasets from real-world projects, such as the NASA Metrics Data Program, and using several regression and classification models (Ostrand, Weyuker, & Bell, 2005), (Menzies, Greenwald, & A.Frank, 2007), (Seliya, Khoshgoftaar, & Hulse, 2010).

In (Chidamber & Kemerer, 1994) object-oriented metrics were proposed as predictors of faults density. A later study (Subramanyam & Krishnan, 2003) empirically validated three OO design metrics suited for their ability to predict software quality in terms of fault-proneness: the Chidamber and Kemerer (CK) metrics, Abreu's metrics for object-oriented design (MOOD), and Bansiya and Davis' quality metrics for object-oriented design (QMOOD). The study presents a survey on eight empirical studies showing that OO metrics are significantly correlated with faults. Further studies on design metrics are in (Binkley &

Schach, 1998), (Ohlsson & Alberg, 1996), where authors investigated the suitability of metrics based on the software design. Basili et *al.* (Basili, Briand, & Melo, 1996) also focused on validating OO design metrics for prediction.

Other studies focused on the definition of software metrics collected from early lifecycle data such as textual requirements: authors in (Jiang, Cukic, & Menzies, 2007) combined requirement metrics with code metrics and reported that requirement metrics improve the performance of models that use code metrics. These metrics have been gathered from textural requirement specification documents by using ARM (automated requirement measurement) tool.

In many cases, common metrics provide good prediction results also across several different products. However, it is difficult to claim that a given regression model or a set of regression models is general enough to be used even with very different products, as also discussed in (Nagappan, Ball, & Zeller, 2006), (Zimmermann, Nagappan, Gall, Giger, & Murphy, 2009). Some works focused on transferring prediction models across different projects and companies, e.g., (Nam, Jialin Pan, & and Kim, 2013). Finally, only few studies considered the problem of discriminating between fault types in fault prediction (Caglayan, Tosun, Miranskyy, Bener, & Ruffolo, 2010), (Mısırlı, Caglayan, Miranskyy, Bener, & Ruffolo, 2011), (Carrozza, Cotroneo, Natella, Pietrantuono, & Russo, 2013), (Cotroneo, Natella, & Pietrantuono, Predicting aging-related bugs using software complexity metrics, 2013).

Although, as described, there are studies using every sort of metrics, authors of (Catal & Diri, 2009) argue that the most used and reliable ones are the method-level metrics, followed by the class-level ones (they count that more than 64% of the works they surveyed adopt method-level metrics), the most common ones being the McCabe's cyclomatic complexity, the lines of code, the Halstead's metrics, and the object-oriented CK metrics.
Thus, a practical way to apply this step is to consider such metrics as a starting point (Table I provides a list used in (Cotroneo, Pietrantuono, & Russo, Testing techniques selection based on ODC fault types and software metrics, 2013), which are more guaranteed by past studies to give good results; other metrics specific to the company or to the product/process features may be then added to refine the prediction accuracy, and their "quality" for prediction purposes iteratively monitored across products.

Table I. A set of commonly used code metrics

Metrics	Description	Metrics	Description
CountDeclClass	Number of classes	MaxNesting	Maximum nesting level of control constructs
CountDeclFunction	Number of Function	CountPath	Number of unique paths though a body of code
CountLine	Number of lines	SumComplexity	Sum of cyclomatic complexity
CountLineBlank	Number of blank lines	SumEssential	Sum of essential complexity
CountLineCode	Number of lines containing source cede	AvgVolume	Average Halstead's volume
CountLineComment	Number of lines containing comments	AvgLength	Average Halstead's Length
CountLineInactive	Number of lines inactive from the view of pre-processor	AvgVocabulary	Average Halstead's Vocabulary
CountStmtDecl	Number of declarative statements	AvgDifficulty	Average Halstead's Difficulty
CountStmtExe	Number of executable statements	AvgEffort	Average Halstead's Effort
RatioCommentToCode	Ratio of the number of code lines to the number of comment lines	AvgBugs	Average Halstead's Bugs Delivered
Header Files	Number of header files	VVolume	Variance of Halstead's Volume
Code File	Number of Code File	VLength	Variance of Halstead's Length
CountLineCodeDecl	Declarative source code	VVocabulary	Variance of Halstead's Vocabulary
CountLineCodeExe	Number of lines containing executable source code	VDifficulty	Variance of Halstead's Difficulty
AvgCyclomatic	Average Cyclomatic Complexity	VEffort	Variance of Halstead's Effort
MaxCyclomatic	Maximum Cyclomatic Complexity	VBugs	Variance of Halstead's Bugs Delviered

6.2 EXTERNAL QUALITY ATTRIBUTES AND METRICS

Besides internal quality of the product, it is also a common practice the measurement of "process" quality, which provide insights on how well the phases of a process work, how people, teams, organizational factors, and all the resources involved in the development impact the final quality, and to what extent all these factors are controllable and thus improvable. All this represents a degree of how much the entire know-how of the company is clearly stated, objectively visible, easily available, and independent from individual skills and competencies.

This Section describes the most common external quality attributes used by practitioners and researchers. The described ISO9126 standard defines a set of 6 external quality, characterized by sub-attributes, which engineers can refer to. However, there exist several different terminologies describing aspects of external quality with slight shades depending on the context and on the attributes of interest. For instance, robustness and reliability are intended in a different way from the dependability community with respect to the software engineering community. ISO 9126 reflects more the latter.
Moreover, the amount of work that has been done to assess or improve the external quality attributes also depend on the easiness in obtaining measurements for that attribute: for instance, "usability" may be more difficult to assess than robustness.
In the following, we briefly describe the most considered quality attributes by both researchers and industrial practitioners, along with a brief mentioning of the quality assurance strategy to assess and improve them.

6.2.1 Dependability and Security

Dependability is defined as the ability to deliver service that can justifiably be trusted. This definition stresses the need for justification of trust. Another way to define it is: the ability to avoid service failures that are more frequent and more severe than is acceptable. This definition stresses more the avoidance of failures (Avizienis, Laprie, Randell, & Landwehr, 2004). Dependability is not a single quality attribute; it is rather viewed as a composition of several attributes, which can be assessed using qualitative or quantitative measures. Avizienis et *al.* define the following dependability attributes:

- ***Availability*** - *readiness for correct service*
- ***Reliability*** - *continuity of correct service*
- ***Safety*** - *absence of catastrophic consequences on the user(s) and the environment*
- ***Integrity*** - *absence of improper system alteration*
- ***Maintainability*** - *ability for a process to undergo modifications and repairs*

Means to attain dependability are classifiable in:

- Fault Prevention, e.g., implementation of good practices of software engineering to prevent fault from being present at operational time; these include as defined requirements engineering processes, good design

principles (e.g., modularity, abstraction, reuse), good coding practices and standard/guidelines compliance, proper V&V processes and techniques, etc..

- Fault Tolerance is concerned with the means to avoid failures by detecting an activated error, tolerate it, and perform system recovery
- Fault Removal means are applied either in the system development phase (through Verification, Diagnosis, and Correction) or at system usage phase, through corrective or preventive maintenance.
- Fault Forecasting focuses on performing an evaluation of the system behavior with respect to fault occurrence of activation; it includes qualitative evaluation (e.g., failure mode analysis) and quantitative evaluation means (e.g., evaluate in terms of probabilities the extent to which some of the attributes are then viewed as measure).

The implementation of these means depend greatly on the attribute being assessed. For instance, taking testing as one of the main assessment techniques, there are many techniques varying according to the attribute under assessment.

Availability and ***Reliability*** are more easily quantifiable as direct measurements, whilst the others are more subjective, and their assessment makes more use of qualitative techniques or mixing quantitative/qualitative techniques.

Differently from availability, which focuses on failures on a given instant of time, reliability emphasizes the occurrence of undesirable events in a specified time interval, often called Mission Time. A lowly available system may be highly reliable in a given mission time, depending on when the failures occur (e.g., a system may fail many times in one year, but never in the time interval that is crucial for system's mission).

Models are extensively used for dependability attributes evaluation, especially for reliability and availability analysis. A first distinction is between combinatorial and state-based models. Contrary to combinatorial models, state-based stochastic ones appear to be adequate to deal with the complexity of the considered class of systems, though they suffer of the state-space explosion problem. This problem has triggered many studies and significant results have been achieved in the last 15 years based on two general approaches: "largeness avoidance" and "largeness tolerance".

Largeness avoidance techniques try to circumvent the generation of large models. They are complemented by largeness tolerance techniques which provide practical modelling support to facilitate the generation and solution of large state-space models. A number of modelling approaches appeared in the literature. They are:

1) compositional modelling approaches (e.g., (J. F. Meyer & W.H.Sander, 1993), (Kanoun, 2004) (Rabah & K. Kanoun, 2003) (Dai, Pan, & Zou, 2007)), both at level of defining suitable composition operators to build models from a set of building blocks, as well as defining composition rules;
2) decomposition/aggregation modelling approaches (e.g., (Bobbio & Trivedi, 1986), (Lollini, Bondavalli, & Giandomenico, 2009), (Ciardo & Trived, 1993)), where the overall model is decoupled in simpler and more tractable sub-

models, and the measures obtained from the solution of the sub-models are then aggregated to compute those concerning the overall model;
3) derivation of dependability models from high-level specification, e.g. from UML design (e.g., (Ganesh & Dugan, 2002)).

A detailed survey of major approaches to modelling based on largeness avoidance and largeness tolerance is in (Nicol, Sanders, & Trivedi, 2004).

However, model-based approaches may be not accurate enough, when the input parameters values are not representative of the real system behaviour. Measurements-based approach may allow for more accurate results: it is based on real operational data (from the system or its prototype) and the usage of statistical inference techniques. It is an attractive option for assessing an existing system or prototype and constitutes an effective way to obtain the detailed characterization of the system behaviour in presence of faults. However, since real data are needed, it is not always possible to apply this approach, because data may be not available. Moreover, just relying on measurement- based approach does not yield insight into the complex dependencies among components and does not allow system analysis from a more general point of view. It is often more convenient to make measurements at the individual component/subsystem level rather than on the system as a whole (Garzia, 2002), and then to combine them in a system model.
An overview of experimental approaches to dependability evaluation is in (Silva & Madeira, 2005). Although the most of papers use either the model based or the measurement based approach, some papers use a combined approach (Tang & Iyer, 1993), (D.Long, A.Muir, & R.Golding). An online monitoring system combining both the approaches towards system availability evaluation is in (Mishra & Trivedi, 2006), (Haberkorn & Trivedi, 2007), (R.Pietrantuono, Russo, & Trivedi, 2010). Models parameterized by experimental data are also used in (Vaidyanathan & Trivedi, 2005) for software aging and rejuvenation analysis (directly related to availability), in which Markov chains are used to model both system states and workload states, and transition probabilities and sojourn time estimates were obtained by using real experimental data.

A class of models that gained importance since the advent of object-oriented and component-based systems are the architecture-based models. These analyze the attribute of interest (reliability or availability, typically) by modelling the relation among components, and how components' interaction impacts on the reliability/availability value attained for the overall system (Gokhale, E.Wong, Horganc, & Trivedi, 2004), (Goseva-Popstojanova, Mathur, & Trivedi, 2001), (Gokhale, Lyu, & Trivedi, 2006), (W.Wang, Y.Wu, & Chen, 2009). This enables evaluating architectural alternatives and supports development in the architectural design stage.
Architecture-based models can be categorized as follows (Goseva-Popstojanova & Trivedi, 2001)

- State-based models use the control flow graph to represent software architecture, modelling the architecture as a Discrete Time Markov Chain (DTMC) a Continuous Time Markov Chain (CTMC) or semi Markov Process (SMP).

- Path-based models compute the system reliability/availability considering the possible execution paths of the program.
- Additive models, where the component reliabilities are modelled by non-homogeneous Poisson process (NHPP) and the system failure intensity is computed as the sum of the individual components failure intensities.

As for reliability, there exist a further class of model-based analysis, known as the "black box" modelling approach. These are aimed to evaluate how reliability improves during testing and varies after delivery, differently from the "architecture-based models", which focus mainly on understanding relationships among system components and their influence on system reliability/availability. The black box approach ignores information about the internal structure of the application and neglects relationships among system components. It is based on (i) collecting failure data during testing, and (ii) calibrating a software reliability growth model (SRGM) using such data. These models, e.g., (A. L. Goel, 1979), (Yamada, Ohba, & Osaki, 1983), (Goel., 1985), (Gokhale & Trivedi, Log-logistic software reliability growth model, 1998), (Mullen, 1998), (Okamura, Dohi, & Osaki, 2004), are then used for prediction in the operational phase (to predict the next failure occurrences based on the trend observed during testing) and/or in the testing phase of successive system releases to determine when to stop testing.

Besides modelling, further assessment methods are through testing. While it is rare to hear about "availability testing", reliability testing dates back to eighties and has a long history. Testing to assess (or in some cases to improve) reliability is often referred to as operational testing. With it, testers aim at achieving high operational reliability, intended as the probability of not failing during the operational phase. Ideally, operational testing is historically considered as the most promising approach at improving reliability, because it is the only testing method devised to find failures with probabilities matching their real runtime occurrence. The term Statistical testing is also used to denote this technique. However, Statistical testing is more recently being used to denote also a slightly different approach whose goal is to satisfy an adequacy criterion expressed in terms of functional or structural properties (e.g., ensuring that each structural element is exercised as frequently as possible (Poulding & Clark, 2010)). The most important contexts in which operational testing has been used are in the frame of the *Cleanroom* approach (Mills, Dyer, & R. C. Linger, 1987), and in the process defined by Musa, i.e., the *Software Reliability Engineering Test* (SRET) practice (Musa, 1996).
More recently, Chen et *al.* (Chen, Kuo, & and H. Liu, 2009), (T.Y.Chen, F.-C.Kuo, & Liu, 2008) tried to improve the effectiveness of uniform random testing by taking account of the feedback information generated by those test cases that do not trigger defects. They use the concept of adaptive random testing (ART) (K.Y.Cai, 2002), looking for test profiles, different from operational profile, able to evenly distribute test cases over the code. Adaptive testing is also the approach adopted in (Cai, Gu, Hu, & Li, 2007). In (Cai, Li, & Liu, 2004), the same approach is used to assess software reliability (i.e., with the objective of minimizing the variance of the reliability estimator). Although the cited approaches promote operational and re- liability-oriented testing, reporting

good results, it is still difficult, at this time, to find compelling evidence that operational testing is always a good practice.

The evaluation of the ***safety*** attribute is not easily done by quantitative approaches. What it is actually done is to combine quantitative with qualitative methods. Some of the most used techniques and principles, also recommended by the mentioned safety-related certification standards (e.g., (CENELEC 50128, 2011), (ISO 26262, 2011) (RTCA - DO 178C, 2011)), are:

- Hazard (and operability) Analysis (HAZOP), risk analysis, failure mode and effect (and criticality) analysis (FME(C)A), fault tree analysis (FTA), formal methods, in the requirement analysis and specification phase;
- Formal methods, modelling, fully defined Interface (e.g., design by contract), temporal/spatial partitioning, design diversity, recovery block, graceful degradation in the design phase;
- Coding standards, coding rules (e.g., no pointer, no global var.), defensive programming, appropriate language (e.g., strongly typed, language subset), for the coding phase;
- Strict V&V Process definition, planning, monitoring and control; functional and structural testing with full coverage, non-functional Testing (performance, robustness,…), static and dynamic analysis, inspection, formal methods, for the V&V phase;
- CVS, defect tracking, data recording and analysis, full traceability, configuration management, design for change, impact analysis.

Typically, standards require several of these techniques according to the safety level that one wants to achieve. The objective of the producer is to provide evidence, through these techniques, that the developed system is safe. Such evidences are often called safety cases.

When we add to these attributes the confidentiality, we also have the attribute of "***security***". Confidentiality is the absence of unauthorized disclosure of information, and the composition of Confidentiality, Integrity, and Availability makes security. Security is sometimes classed as an attribute (Sommerville, 2004) but the current view is to aggregate it together with dependability and treat Dependability as a composite term called Dependability and Security (Avizienis, Laprie, Randell, & Landwehr, 2004).

Security is huge field of research, ranging from network-related security, to cryptography, from physical (transmission)-level to protocol-level, up to application-level security. From the software perspective, works regarding the software design and testing for security are more of interest. A number of papers regard design for security (Laverdiere, Mourad, Hanna, & Debbabi, 2006), security models (e.g., (Jing, 2010) (Roy, Kim, & Trivedi, 2012)) formal definition of security attributes and metrics (e.g., (Krautsevich, Martinelli, & Yautsiukhin., 2010)), testing and analysis for security (see (AL-Ghamdi, 2013)). In the practice, the most used approaches to assess security are testing and analysis techniques. They include: code reviews, automated static analysis, binary code analysis, fuzz testing, source and binary code fault injection, risk analysis, vulnerability scanning, penetration testing.

The last attribute of dependability is ***maintainability***. The definition of maintainability as per IEEE is defined as:
"The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment is maintainability". There are four major categories of maintenance (Lientz & Swanson, 2000):

- Corrective maintenance: Reactive modification of a software product performed after delivery to correct discovered problems.
- Adaptive maintenance: Modification of a software product performed after delivery to keep a software product usable in a changed or changing environment.
- Perfective maintenance: Modification of a software product after delivery to improve performance or maintainability.

Maintainability models and techniques can be divided roughly in two classes (Tiwari & Sharma, 2012):

- *Traditional Maintainability Technique*: these include works defining models that include several attributes. For instance (Khairuddin & Elizabeth, 1996) includes attributes as: Modularity, Readability, Programming Language, Standardization, Level of Validation and Testing, Complexity and Traceability used to assess maintainability; (Fioravanti & Nesi, 2001) includes considers adaptive maintenance, defining a proper metric; Bandini et al., (Bandini, Paoli, Manzoni, & Mereghetti, 2002) considered three independent factors, namely; design complexity, maintenance task and programmer's ability to predict the maintenance performance for object-oriented systems; Ahn et al. (Ahn, Suh, Kim, & Kim, 2003) proposed a software maintenance project effort estimation model, which is based on the function point measure and 10 maintenance productivity factors;
- *Soft Computing based Maintainability Techniques:* include more recent techniques to assess maintainability based on a set of predictor factors: these can be grouped in techniques based on aritifical neural network, Adaptive Neuro Fuzzy Inference System, Fuzzy approach, Genetic Algorithm (Singh, Kaur, & Sangwan, 2004), (Aggarwal, Singh, Chandra, & Puri, 2005), (Aggarwal, Singh, Kaur, & Malhotra, 2006), (Shukla & Mishra, 2008).

6.2.2 Performance

Performance is an external attribute based on user requirements, related to the ability of the system to provide the required response within time and resource constraints.
Performance is mainly assessed by either modelling approaches (e.g., in the design phases) or by performance testing techniques. Performance modelling techniques are similar to the ones used for reliability and availability analysis, being them based on Markov-like formalisms and network queues, aimed at analyzing the possible behaviours of a system from the architectural point o

view, by changing parameters of interest and improve planning and resource allocation (Pietrantuono, Russo, & Trivedi, 2010).
On the other hand, a set of approaches focuses on testing to assess and then improve performance. In fact, the predecessor of performance problems is the absence of test planning (Weyuker & Vokolos, 2000); hence, strategies that support the choice and the execution of performance testing are fundamental. In this direction (D. G. & Emmerich, 2004), the authors underline the importance of designing and executing performance testing since early architectural design phases. They propose an approach that supports the selection of relevant use cases from the architecture design and the execution of such tests using early available software. Liu et *al.* propose in (Liu, Gorton, Liu, Jiang, & Chen, 2002) a hybrid approach based on empirical testing and a queuing network modelling to predict the performance of component-based applications. They point out the importance to isolate performance issues due to the business logic from the ones due to the underlying middle-ware infrastructure (i.e., the container), and to the operating environment (e.g., hardware, OS, and DBMS). In (A.Avritzer & E.J.Weyuker, 2004) the authors combine performance testing and simulation model to predict performance problems of an e-commerce application, consisting of a front-end web application, a middle-tier application server, and a back-end database. The paper also provides a series of steps to support the diagnosis of performance slowdown, as well as a procedure for automatic test case generation and execution. Other approaches based on testing that evaluate the performance of component-based systems are surveyed in (Koziolek, 2010).The authors classify approaches according to the development cycle phase in which they are applied, and then discuss their benefits and drawbacks. An earlier study (Becker, Grunske, Mirandola, & Overhage, 2006) also surveys model- and measurement-based approaches that address performance assessment of component-based systems.

6.2.3 Robustness

Robustness is defined as the degree at which a system operates correctly in the presence of exceptional inputs or stressful environmental conditions (IEEE 610.12).
As the robustness of a software component has repercussions on the dependability of the whole system, its correct evaluation is essential. It is particularly relevant software Components taken Off-The-Shelf (COTS), because they have been developed by ignoring a specific application context.
A common way to assess robustness of the software is through robustness testing. It aims to evaluate the capacity of a system to resist and react to exceptional and erroneous inputs. Robustness testing was introduced as an automatic technique treating the tested system as a black-box. Robustness testing campaigns based on robustness failure rate (the percentage of non-properly handled erroneous inputs) allow identifying several improperly handled inputs.

Robustness is evaluated in presence of the erroneous inputs at the interface of a component/system. Such erroneous inputs may originate from faults in some other components, or from human mistakes in providing inputs to the interface.
Based on this view, there are roughly two approaches to robustness testing. In the former, faults may be injected into a component A and, if and when they become active, there is a chance that they manifest at the interface of the component A as errors: then these errors may propagate to component B which is the target of the robustness testing. The injection of faults in A is achieved through what is known as code mutation. This technique has been adopted in (Duraes & Madeira, 2002); although it is effective, it requires that injected faults are representative of real faults, and of their activation and their propagation to component B's interface.
The latter approach is to inject errors directly at the interface of the target. In practice, the parameters of the service (a function) are corrupted with specific errors. This approach compared to the former one does not require the activation of a fault, but it is sufficient to observe a service invocation towards the target. In both cases, the service interface is the error location. For this reason robustness testing is also named interface error injection (Natella, 2011).
Robustness testing has been widely and successfully applied to the OS system call interfaces or device drivers (Johansson, Suri, & Murphy, On the impact of injection triggers for OS robustness evaluation, 2007) (Sarbu, Johansson, Suri, & Nagappan, 2009). These interfaces are of interests because through them it is possible to assess the robustness of the OS against erroneous behavior of applications and drivers, which have been proven to be particularly error prone (Chou, Yang, Chelf, Hallem, & Engler, 2001). The types of errors injected at the service interface can be classified according to three types of error model (Johansson, Suri, & Murphy, On the selection of error model(s) for OS robustness evaluation, 2007):

- *Fuzzy*; errors are chosen randomly among all possible values of the input domain of the service. Therefore experiments with this type of errors should be repeated a significant number of times to be confident in the final results.
- *Data Error;* errors are selected according to the type of the input parameters. The selection of the error is conducted on the basis of the tester experience or with established methods (e.g., boundary analysis). Depending on the type of parameters, the number of injections can vary from one case to tens of cases for a given parameter (Johansson, Suri, & Murphy, On the selection of error model(s) for OS robustness evaluation, 2007).
- *Bit Flip*; errors are permutations of one of the bit of the input parameter of the service. This model derives from hardware errors in which the real faults are modelled as "bit flips" (Hsueh, Tsai, & Iyer, 1997). It is easy to use, but it requires many experiments because of different number of the bit to flip for all the input parameters.

The errors can be injected in a precise temporal interval, time-driven injection, or when specific events occur, event-driven injection. The time for the injection, often, is not precisely defined, rather it is assumed that the system is in a given state when executing the RT. The event-driven approach injects errors when a precise sequence of calls to the service interface takes place.

Some of the most successful approaches, with the associated tool, is Ballista (Koopman & DeVale, The exception handling effectiveness of POSIX operating systems, 2002), (Koopman, Sung, Dingman, Siewiorek, & Marz, 1997). It was the first approach for evaluating and benchmarking the robustness of commercial OSs with respect to the POSIX system call interface (IEEE Std 1003.1b, 1993). BALLISTA adopts a data-type based fault model, that is, it defines a subset of invalid values for every data type encompassed by the POSIX standard. A test case consists of a small program that invokes the target system call using a combination of input values. Test outcomes are classified by severity according to the CRASH scale: a Catastrophic failure occurs when the failure affects more than one task or the OS itself; Restart or Abort failures occur when the task launched by BALLISTA is killed by the OS or stalled; Silent or Hindering failures occur when the system call does not return an error code, or returns a wrong error code. BALLISTA found several invalid inputs not gracefully handled (Restarts and Aborts), and some Catastrophic failures related to illegal pointer values, numeric overflows, and end-of-file overruns (Koopman & DeVale, The exception handling effectiveness of POSIX operating systems, 2002).

6.2.4 Usability

Usability is the ability of a software system to be comprehended, learned, used with satisfaction by users in specified usage conditions. It means that software manages well the interaction with users. At the beginning, the software engineering community has always associated usability with interface design. Then several compsite models have been developed.

There are older quality models including usability in their definition, such as the Boehm Model (1978), and the Mc Call Model (1977) - also called GE model or FCM (factor, criteria and metric). However, these older models do not capture our current meaning of usability.

In the literature, one of the first authors in the field to recognize the importance of usability engineering was Shackel (Shackel, 1991). In his approach, Shackel defines a model where product acceptance is the highest concept. The user has to make a trade-off between utility, the match between user needs and functionality, usability, ability to utilize functionality in practice and likeability, affective evaluation versus costs; financial costs as well as social and organizational consequences when buying a product. Usability is defined as: ‘‘the usability of a system is the capability in human functional terms to be used easily and effectively by the specified range of users, given specified training and user support, to fulfil the specified range of tasks, within the specified range of scenarios’’.

For a system to be usable it has to achieve defined levels on the following scales:

- Effectiveness: performance in accomplishment of tasks.
- Learnability: degree of learning to accomplish tasks.
- Flexibility: adaptation to variation in tasks.
- Attitude: user satisfaction with the system.

Nielsen (Nielsen, 1993) follows Shackel, considering usability as an aspect that influences product acceptance. Acceptability is differentiated into practical and social acceptability. It considers usability as composed of:

Learnability; Efficiency (systems should be efficient to use), memorability (systems should be easy to remember), errors (the system should have a low error rate); satisfaction (the system should be pleasant to use).

The ISO 9126 standard currently defines it as "The extent to which a product can be used by a specified set of users to achieve specified goals (tasks) with effectiveness, efficiency and satisfaction in a specified context of use". It is viewed as made up of: understandability, learnability, operability, attractiveness (i.e., user-friendliness).
Authors in (Folmer & Bosch, 2004) surveys these definitions, and ends up by dividing usability attributes in:
Objective operational criteria: user performance attributes such as efficiency and learnability.
Subjective operational criteria: user view attributes such as satisfaction and attractiveness. Of course subjective criteria are more difficult to assess.

As for the evaluation, Zhang (Zhang, 2001) has identified three types of usability evaluation methods: *Testing, Inspection, Inquiry*.
The usability testing approach requires representative users to work on typical tasks using the system or the prototype. The evaluators use the results of testing to see how the user interface supports the users to do their tasks. Testing methods include the following (from (Folmer & Bosch, 2004)):

- Coaching method
- Co-discovery learning
- Performance measurement
- Question-asking protocol
- Remote testing
- Retrospective testing
- Teaching method
- Thinking aloud protocol

The usability inspection requires usability specialists or software developers, users and other professionals to examine and judge whether each element of a user interface or prototype follows established usability principles. Commonly used inspection methods are: Heuristic evaluation; Cognitive walkthrough; Feature inspection; Pluralistic walkthrough; Perspective-based inspection; Standards inspection/guideline checklists.

Usability inquiry requires evaluators to obtain information about users likes, dislikes, needs and understanding of the system. Inquiry methods include: Field observation. Interviews/focus groups; Surveys; Logging actual use; Proactive field study; and Questionnaires. About the latter, (Zhang, 2001) and various other web resources provide an overview of web based interface evaluation questionnaires.

7 FACTORS IMPACTING QUALITY AND COST

Besides quality characteristics of a product, those factors that most prominently impact the quality need to be reviewed. In fact, the ICEBERG project focuses on determining a model-based process for linking (poor) quality attributes with cost. External factors impacting such quality are therefore to be considered in subsequent steps.

Main objectives of projects are defined by the classical model of restrictions of the triangle of project management. This model also knows as the iron triangle (see Fig. 3) was invented by Dr Martin Barnes in 1969 to demonstrate the connection between time, cost and output (correct scope at the correct quality) (APM, 2010).

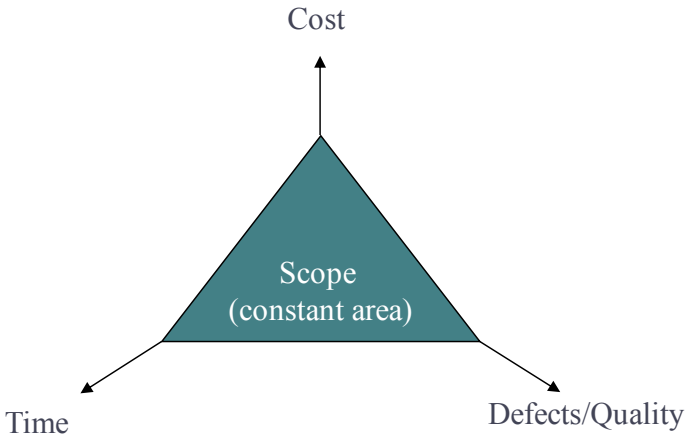

Fig. 3 The project management triangle or iron triangle

Keeping in mind the interest in the three dimensions which project managers want to control, we want to analyse which are the sources for quality while looking at the necessary effects in the two other dimensions: cost and time. This idea is expressed in this document as the factors that impact quality, cost and time. Although many classifications might be adopted, the model presented by (McConnel, 1996) would help us to allocate current lines of actions in big categories according to their core philosophy. The model proposes four main classes of factors influencing results in software projects (see Fig. 4).

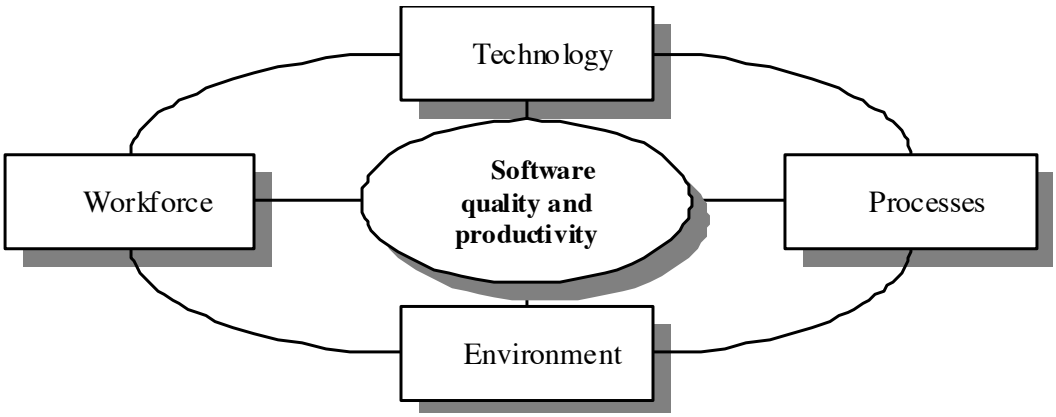

Fig. 4 Factors for software quality and productivity

In the subsequent sections, we will review some of the relevant existing references in the literature that describe the factors of each type and how they impact results.

7.1 THE IMPACT OF THE PROCESS

The category of processes is huge as it embraces both processes in the large (with a special focus on quality assurance and management activities) and specific methods and techniques for detailed tasks at project level (also with specific interest on quality-related activities). In the first group, the main effort has been devoted to the well-known process models like CMMi (CMMI Product Team, 2006) and ISO15504 (ISO 15504-1, 2004). Lastly agile methods such as SCRUM (Cohn, 2009) have been emerged with energy as proposed solutions for quality and productivity. The more traditional proposals like Unified Process (Jacobson, Booch, & Rumbaugh, 1999) or classical methodologies and life cycles models are not today choices if not embedded within bigger frameworks, especially because they have been well accepted but almost no real rigorous data on benefits have been provided.

At least, different studies on CMMi influence have been published during the last years, e.g. (Goldenson & Gibson, 2003) and (Herbsleb, Carleton, Rozum, Siegel, & Zubrow, 1994). Within the proposals of processes in the large, the specific proposals centred on quality assurance are also important chances, specially the adaptation of the general ISO 9001 (ISO 9001, 2008) to software development (ISO/IEC 90003, 2004). Although its implementation in the software companies has been scarce and with a total absence of measurement of benefits, it is true that its consequences in organizations (systematic documented processes, etc.) represent a minimum application of quality practices that provide some trust in reaching project objectives.

In the category of processes at project level, specific quality plans, for example (IEEE, 1984), usually rely on a traditional set of techniques to help in quality assurance: configuration management, metrics/measurement, verification and validation (highlighting software testing and review and audits processes as commonest methods although additional techniques are available but not very

common (Adrion, Branstad, & Cherniavsky, 1982)). It is important to remark that these techniques require non trivial investment in human effort (hence money), so their use should be adapted to each organization both to be effective (e.g. providing an easy-to-use environment) and to be efficient and cost-effective (e.g. reviewing all the code is usually unfeasible; so a previous selection of defect-prone parts using metrics and Pareto principle and the help of automatic reviewing tools are a must). In general, adequate quality assurance efforts tend to make more stable delivery schedules and quality levels (Herbsleb, Carleton, Rozum, Siegel, & Zubrow, 1994). A good number of studies (e.g. (van Solingen, 2006) and (Rico, 2004)) show productivity improvement and good ROI based on decreasing rework and bad quality costs (Fleckenstein, 1983), especially when focus is put on prevention and early detection because fixing cost rises sharply when defects remain undetected throughout the project phases (Harrington, 1987).

Processes are not equally implemented in all types of settings, and these differences lead to different decisions in projects and different effects in results and costs. This is the case of SME (Small Medium-size Enterprises). Empirical studies state that there exist a lot of problems in small and medium scale companies, which must be taken into consideration while talking about the quality in software product. An example is described by (Pusatli & Misra, 2011) with empirical investigations for quality assurance activities in SME in Turkey. Due to constraints in resources and business model, management tend to work with some thoughts related to quality: quality issues are not the main goal during projects, large standards like ISO or CMMI are mainly considered for external image, there is a lack of expertise and customer satisfaction which is the only priority does not need certifications and complex QA.

Another scenario where decisions on quality and related costs are also affected by human factors is global software development (GSD) as described by (Misra & Fernández-Sanz, 2011). A significant number of studies are available in literature analysing the problems of developing software in distributed environments. The processes normally have to change to cope with the clear limitations imposed by these projects. Although many of the factors in GSD (cultural differences, geographical distance, time difference, etc.) can better be included in the category of environment (see Section 7.2) or workforce (see Section 7.3), it is clear that processes have to be adapted and that effects of decisions on quality and costs are different.

Other scenarios and settings for projects lead to differences in the application of QA techniques and methods or in the selection of the tools, methodologies or techniques that could be more appropriate for assuring quality results while controlling they are effective and efficient in costs and time. Even selection of processes in the large through CMMi or ISO models or adoption of life cycles (incremental, iterative, spiral cycles, etc.) and development methodologies or philosophies (agile, standard methods, etc.) is not leading to totally homogeneous ways to work in projects: on the contrary, they promote the necessary customization to company features and circumstances. In fact, each project is going to adapt general guidelines through decisions of project

managers in terms of process adaptation, allocation of resources, selection of tools and techniques, etc.

7.2 THE IMPACT OF THE ENVIRONMENT

Environment and personnel are the two factors which are by far the least analyzed in a formal way, specially their relation to effects on software quality and productivity described in a measurable manner (Thomas, Hurley, & J.:, 1996). As development cost models have shown, humans represent the main resource (and the main cost) for software projects. In general, proposals related to cost estimation and project management have been the pioneers in the analysis of human and environmental factors due to their influence on budget, schedule and quality. One sophisticated example of modelling of relations among the different factors which influence software development projects in effort and results during is the project dynamics proposed by (Abdel-Hamid & Madnick, 1991). But people are not isolated from their professional environment: office and working conditions as well as psychological aspects are influencing their productivity and the quality they produce. Although studies and models clearly describing how the environment affects results in projects are scarce, at least we can mention some remarkable results from them. The work in (Jones, Estimating software costs, 1998) determines with the reference to function points statistics that ergonomic offices can add an average 15% of productivity to development while crowded and uncomfortable ones deduct 27% and that low moral decreases in an average 6% productivity while high moral and good mood may add a 7%. Even decisions of managers which impact the psychological conditions of professionals would lead to clear effects: a moderate schedule pressure gets an additional 11% of improvement in productivity while an excessive one provokes a radical decrement of 30%. In specific settings like geographically distributed projects and GSD, cultural differences usually create communication barriers and problems of coordination in the team. As described by (Misra & Fernández-Sanz, 2011), they may also affect productivity and quality of by results foreigners who work during a period in another country and even their acceptance by local team. Time differences between countries in GSD projects is another important environmental factor. These reasons are causing a restructuration of outsourcing destinations all over the world promoting the change to nearshoring.

7.3 THE IMPACT OF THE WORKFORCE (HUMAN FACTORS)

As stated in previous section, workforce is the main resource of a software project and it accounts for the most of the cost. Together with other factors, the project dynamics model of (Abdel-Hamid & Madnick, 1991) is one of the most complete to show all factors and their consequences in projects through a complex network of influences. Some subsequent efforts on this model have checked if the famous Brook's law, "adding people to a delayed project tends to delay it more and with higher cost" (Brooks, 1975), is confirmed with the model: one study confirms higher costs but not always more delay (only when tasks are highly sequential) (Hsia, Hsu, & Kung, 1999), others restrict the effects to small projects below 5 people involved. Of course, this can be pointed out using the traditional study of Schulmeyer in which the phenomenon of the net negative producing programmer is exposed (Schulmeyer, 1992).

However, locating quantified analysis of development personnel is usually linked to contribution in the area of software cost estimation models. An evident reference in this approach is the quantification of influence of qualification and experience of development personnel in the traditional COCOMO drivers (van Solingen, 2006): analysts' capacity, experience with similar applications, programmer's capacity, experience with virtual machine and experience with language. Letting aside the precision and validity of COCOMO, it is interesting to realize that negative influence of these factors is always greater than the corresponding positive influence when having better than average situations (e.g. low valued analysts represent 46% of extra cost while highly skilled analysts contribute with a 29% reduction of costs).

Another interesting set of data on the influence of factors related to software development personnel was published by C. Jones (Jones, Estimating software costs, 1998) within his wide statistics on software costs, mainly based on function points. Table II extracted from (Jones, Estimating software costs, 1998) shows the different influence of key factors when situation is positive (e.g., very experienced personnel: +55%) and when it is negative (e.g., -87%). In general, accumulation of negative factors in environment and personnel means a big fall of productivity in much higher degree than positive factors. As a consequence, the sentence "people are our main asset" is even more justified in software development, at least avoiding not cutting costs too much in this aspect.

Table II. Impact of key factors affecting productivity ((Jones, Estimating software costs, 1998)

Factor	Positive influence (+%)	Negative influence (-%)
Lack of experience in managers	High +65%	Low –90%
Developers' experience	High +55%	Low –87%
Not paid extra hours	Yes +15%	No 0%
Annual training	>10 days +8%	No training -12%
Organization	Hierarchical +5%	Matrix –8%

As seen, moral and motivation is a key factor. Research into motivation of software developers suggests they might resist the implementation of quality management techniques (Cheney, 1984) (Couger & Zawacki, 1980) (Woodruff, 1980). In general, there are many sources of motivation and demotivation for software engineers (Beecham, Baddoo, Hall, Robinson, & Sharp, 2008) and motivation is recognized as a major influence for quality (Thomas, Hurley, & J.:, 1996) with many studies analysing level of motivation of software developers (compared to other professions or positions) and their resistance to change (really high in developers engaged in software maintenance (Griesser, 1993)) but with a favourable influence of human relations on motivation (Guterl, 1984). In fact, the use of disciplined teams for development (with disciplined methods) tends to produce better results than ad-hoc or individual programmers (Basili & Reiter R. W., 1979). This is also aligned to models more oriented to personal performance of developers like PSP. Given that one important de-motivator is producing poor quality software with no sense of accomplishment (Beecham,

Baddoo, Hall, Robinson, & Sharp, 2008), total freedom or ad-hoc arrangements would be not a real motivation for software professionals.

When dealing with software quality and reliability, human factors have been identified as a cause of problems that can determine success or not of a project or system (Ren-zuo, Ruo-feng, L., & Zhong-wei, 2004). As stated in several articles (Bach, Enough About Process: What We Need Are Heroes, 1995) (Bach, What Software Reality Is Really About, 1999), the quality of the people should be considered the primary driver for software quality while sometimes too much industry focus has been on the process ("[software] It's more about people working together than it is about defined processes"). From an intuitive and experiential perspective, the education and abilities of a developer represents an important part in the ultimate quality of their developed software but little empirical evidence to support this logical assumption. One of the scarce studies analyzing this relationship has shown that higher proportions of skilled engineers had the most dramatic effect in terms of adequacy of the design and implementation while higher proportions of less skilled engineers negatively affected the end product quality of the software (Beaver & Schiavone, 2006). However, the study allocates the most dramatic positive effect to functional completeness to experienced leader-ship in all stages of the project. Some additional experiments (Zuser & Grechenig, 2003) (Ting-Peng, Jiang, Klein, & Liu, 2010) (Hoegl & Gemuenden, 2001) (Wong & Bhatti, 2009) tend to show a correlation between good teamwork dynamics or other indicators (e.g. size between 5 and 7) and success and quality in the corresponding projects. Team dynamics might be affected by cultural difference as described by (Hofstede, 2001)) although differences are clearer in certain aspects of work as shown by (Fernández & Misra, Analysis of cultural and gender influences on teamwork performance for software requirements analysis in multinational environments, 2012)

It is clear that attitude is an important factor for both quality and productivity that might be also linked to professional ethics codes (Fernández & García, Software engineering professionalism, 2003). Especially attitude of testers and developers towards testing is considered critical for software quality (Murugesan, 1994) (Acuña, Gómez, & Juristo, 2008) (Gill, 2005).

Obviously one key point is the adequacy of education, qualification and soft skills to the corresponding position. It is difficult to have clear definitions of requirements and skills needed for each role or position. Although big efforts on collecting information from, e.g., job ads enable certain general descriptions (Fernández, Personal Skills for Computing Professionals, 2009), it is hard to quantify qualification and experience of people in order to find out possible relationships to effects in productivity and quality with enough accuracy. However, as stated in data included in Table II, training is one of the key factors which can be evaluated by collecting data from professionals as well as the evaluation of developers in terms of factors which influence and their perception of their own skills.

7.4 THE IMPACT OF THE TECHNOLOGY

It is supposed that technology improvement and evolution improves the productivity and quality. This is an evident option for almost all professionals, sometimes boosted by the commercial activity behind each new technical option launched to market. Operating systems, platforms, languages, etc. in a general view but also improved functionality for developers' tools (IDE, CASE, etc.) and new architectures and paradigms. Sadly, it is usually hard to find out independent and rigorous studies with data from real practice which support performance improvements proclaimed by vendors (maybe because they have not been measured or due to hidden difficulties). In general, better tools tend to increase productivity: it is supposed that advanced tools could lead to –17% of effort while extremely basic tools would cause a +24% (Boehm, 1981). In any case, true potential for this line of action is heavily linked to real use by practitioners, who should be promoted by implementing a formal and customized process of acquisition, implementation and training.

Sadly, organizations tend to be more conscious in purchasing in tools (more than 50% according to (McConnel, 1996) than in investing in other actions which might be more efficient. Studies show that effective CASE approaches increase productivity by 27% but an inadequate implantation would lead to a fall of 75% and it is remarked the danger of exaggerated and sensational ads, that 10% of them are purchased but never used, that 25% are poorly exploited due to lack of training, etc. has shown that 75% of statements in ads are considered as ones with low credibility although, after reviewing more than 4,000 projects, 70% of people in charge of projects believe that one unique factor like this would provide big improvement. In fact, many managers still look for the new best tool, language or platform which as a silver bullet will kill all the problems, even if they are not even considering that getting results depend on an effective implementation within the daily practice, changing methods and processes and providing training to soften the learning curve.
Technology may be also present as an influence for project decisions when stakeholders require a specific technical scenario as target operative environment. Software cost estimation models and methods have tried to determine how this type of requirements influence the costs and the complexity of projects, especially when dealing with development languages: experience in using the target language is a cost driver which was already included in the first of COCOMO (Boehm, 1981).

8 QUALITY-BASED DECISION-MAKING PROCESS

"There are known knowns; there are things we know that we know.
There are known unknowns; that is to say, there are things that we now know we don't know.
But there are also unknown unknowns – there are things we do not know we don't know."
(Rumsfeld, 2002)

8.1 INTRODUCTION

This Section is concerned with the analysis of the decision-making processes from an industrial viewpoint. From such processes, the industrial requirements for implementing a cost-effective software quality assurance will be collected and formalized.

Knowing how to make decisions is critical in business management; through this process, the managers of functional areas can determine the type and content of actions and, therefore, their "results".

The decision-making process is critical to any organization that aims to improve efficiency and customer perception. Especially complex organizations should have effective tools for decision-making so that the choices can be carried out quickly and for every level of society.

Over the past 50 years, different models for decision making have been developed. Each of them considers a different "rationality" and a different combination of the variables involved in the process. They develop a thought according to which organizations are mechanistic systems, regulations and rational, and their path is focused on maximum efficiency. The limitations arising from the individual rationality are exceeded by the capacity of organizations in being able to make decisions (Simon, 1960).

The neoclassical view initially saw the man with a full rationality; he was able to choose a better alternative among all the possible ones, with the logic of maximizing results and cost / benefit. However, later, a new theory has been developed in which man has no full rationality and is unpredictable in his actions; his ability to process information is also limited because it depends on his knowledge and experience which is always fragmentary (many times there are things we do not know we don't know). The following Table shows some elements involved in a decision-making process and the different philosophies.

Table III. The main elements of a decision-making process

Objective rationality	Limited rationality
All alternatives of action are known	Not all the alternatives of action are known
It is possible to calculate all the consequences of each action	It is not possible to calculate all the consequences of action: knowledge is

	fragmented
The information are a free commodity	The information are expensive
The management has an exact utility function of its current and future choices	The preferences are not perfectly ordered and their change over time is not predictable
The decision maker is the only one	The decision makers are more than one
The decision is based on optimizing calculation	The decision is based on heuristic vision
The choice is a synoptic process	The choice is a sequential process

The rationality would lead to the choice of an alternative among all those available ones, but the human mind has some alternatives leading to decisions not valid in the context in which it operates. Decisions may be irrational because of:

- incompleteness of knowledge;
- difficulty of predicting;
- variety and variability of behavior;
- adaptability and flexibility based on experience or on communication;
- memory;
- attention and the habit.

Do not neglect the objective, which become part of the elements involved in the decision-making process because it mainly drives the choices. The desire to achieve the final objectives leads to questionable choices; it is not, in fact, applied a fair criterion, and the manager proceeds without considering the different aspects. From another point of view, the objectives do not represent the absolute purpose in view of objectives furthest. This creates a sort of hierarchy between the "means" and "ends" that are distributed along a chain. The relationships between them are never entirely clear and explainable, and also other unconscious, not easily detectable, elements could be involved.

It would be also right to consider the decisions of others as a variable to keep in mind before making choices. Human action is intentional, but also limited, rational. The decision maker tries to be up to the task of finding the optimal solution, but has limitations in terms of decision-making powers. He then adopts a process of research and satisficing, which leads him to look for a pretty good solution.

The criteria to perform this function in the decision-making process, in which the complexity and uncertainty make it impossible global rationality, are:

- *Approach satisfying* (simplification of research, keeping the complexity of the real situation, retaining all the details, limiting the alternatives evaluated by setting a level of suction and try to find a satisfactory alternative in relation to expectations).

- *Approximate optimization approach* (it begins with a description of the situation and to simplify the real one, continuing to reduce it to a level of less complexity, so that the decision maker can contemplate and manage it properly. All inside of this simplification identifies the solution. This is the typical approach of quantitative systems).

8.2 THE EIGHT STEPS FOR A DECISION MAKING PROCESS

The rational approach to individual decision emphasizes the need for systematic analysis of the problem, followed by a choice and realization, along a precise logical sequence. A deep knowledge of the process of rational decision-making can help managers make better decisions even in the absence of clear information. About rational approach, the decision-making process can be divided into eight steps; the first four belong to the stage of identification of the problem, while the others represent the stage of solving the problem of decision-making (Montanari, 2005):

1. *Monitor the surrounding environment*: the manager collects and evaluates internal and external information to identify any deviations from the original objectives.

2. *Define the problem*: the manager identifies the main causes and all the essential elements related to the problem and he assesses its impact on business.

3. *Specify the objectives of the decision*: the manager defines the intent and expectations so the results to be obtained in relation to a decision.

4. *Diagnosing the problem*: the manager tries to understand the cause of the problem in order to respond with appropriate actions.

5. *Develop alternative solutions*: the manager creates a plan of action; the different options need to be easily interpreted for a choice in line with the initial objectives. The decision-making team proposes alternatives that present the highest possible requirements and comply with the highest number of goals. Usually, the solutions differ in their ability to meet the objectives set. The solutions that do not meet the objectives must be immediately removed and not considered further.

6. *Evaluate alternatives*: using statistical techniques or personal experiences to assess the likelihood of success; the choice of method should be made considering also the complexity of the problem.

7. *Choose the best alternative*: The manager uses the analysis of the problem previously performed, without neglecting objectives and alternatives to select the alternative with greater chance of success.

8. *Realize the alternative choice*: once you have chosen an alternative, the manager uses his management skills, administrative and his persuasion skills to ensure that the decision actually solves the problem. We analyze deviations from the objectives and requirements considering the indicators of decision makers.

8.3 THE FOURTEEN FACTORS FOR THE SUCCESS

Using a valid decision-making approach is helpful in any situation, even under conditions of uncertainty, because these steps guide to make a decision that is compatible with the preferences and behavior of an individual (Ferrari, 2010). Fourteen factors that contribute to the success of the decision-making process can be identified as:

1. management commitment
2. perceived benefits
3. quality of management decisions
4. user involvement in the design and implementation of the system
5. commitment of the user to the system
6. costs
7. usability of the system
8. functionality of the system
9. user's knowledge
10. training
11. adequacy of the relationship between the system and the context in which they must provide their support
12. level of use of the system
13. technology used
14. quality of system

The ***management commitment*** (factor 1) is critical to the success of an information system; it depends on the perception of the benefits that can be gained and the costs that represent a limit to the decision to invest in information systems. The commitment increases when increasing the perceived benefits and decreases when the costs are higher.

The real benefits provided by a system often do not coincide with the ***perceived benefits*** (factor 2) from users and managers. The real benefits, in fact, are not easily identified and measured. The commitment of management and user system includes an improvement of the quality of information to support decision-making, therefore an improvement in the quality of decisions, increased productivity and greater efficiency in management and organizational effectiveness. The difficulty in achieving performance is also determined by the complexity of the issues that the business intelligence system is intended to

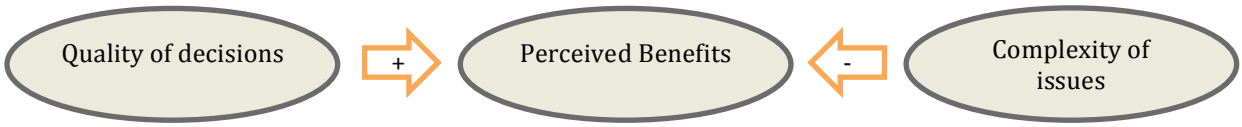

Figure 5 - The perceived benefits directly dependent on the commitment of management and users, and the complexity of the issues, but with the opposite sig4n

solve, understood as the sum of some important elements (such as the number of variables that are the basis of a decision, their degree of interaction, the level of uncertainty associated with the precision with which each variable can be

measured, the time needed to make the decision, the decision-maker's mental and cognitive map (Figure 5)).

In relation to the degree of complexity, there are two types of organizational decisions:

- **programmed decisions** are employed for known problems and routine in which the decision alternatives are clear and their consequences are predictable with a high degree of accuracy. Decisions are very often used in planned or rational decision-making criteria derived from experience, so the alternative choice will be a success.
- **unplanned decisions** concern issues that the organization cannot clearly outline, in which the decision alternatives are uncertain and for which the accumulated experience or rational decision-making methods are found ineffective.

A business intelligence system has as its primary objective in the improvement the effectiveness of decision-making; but how is it possible to better assess if you have achieved your goal? Several aspects are involved: the organizational environment, the approach to the use of the system, the difficulty in understanding the decision-making processes.

8.3.1 The effectiveness decision and the level of use of the system

The ***quality of management decisions*** (factor 3), index of effectiveness of the decision-making process, depends on the complexity in inverse proportion, the level of use of the system, and the quality of the system itself in direct proportion. The ***user involvement*** **in the design and implementation of the system** (factor 4) is a key factor for the development of a decision-making process. In fact, as the perceived utility of the system increases, it allows the user to experience a positive effect and to have a more sensitive on the perception of the quality management system. The ***commitment of the users*** to the system (factor 5), positively relates to the knowledge domain of reference, the perception of the benefits arising from its use and its usability. An indirect relationship is defined, however, with the costs associated with the implementation of the system. The ***costs*** of a BI system (factor 6) relate to technology, investment in human resources involved in the development of the system, training of people who will use the system. They are influenced, so directed, by the level of use of the system. The quality of interaction between the user and the decision support system refers to the concept of ***usability of the system*** (factor 7): it depends, mainly, by the ease of user-system dialogue. The elements involved are definitely the technology used, and the knowledge and expertise of the user. About the ***functionality of a system*** (factor 8) intervening variables that result directly on the type and scope of the technology that is the basis of the system. The ability to not only refer to the characteristics of the system in terms of data access and analysis tools, but also the ability to provide appropriate means to the decision maker to address the various stages of the decision-making process, and the flexibility to the user to analyze the problem, define the alternative, dispute resolution and to choose a plan of action.

A system of BI should direct users to the correct understanding of the problems and improving their skills and ***knowledge*** (factor 9) making use also of the experience. So you gain some awareness, influenced by their knowledge of the system and its level of use.

8.3.2 The importance of training

The user knows how to use the system with wing ***formation*** (factor 10), which allows to expand the knowledge. The training does not neglect the complexity of the application and the user's familiarity and competence in similar applications. The user will then be more inclined to increase the level of acceptance of the system. This factor does not refer only to the technology, but it also suggests an approach in solving the problem, and then, in the decision-making process.

It also requires attention to the ***adequacy of the relationship between the system and the context in which it provides support*** (factor 11). Failure often leads to not use the system; you should take into account the system's functionality, complexity and user involvement in the development.

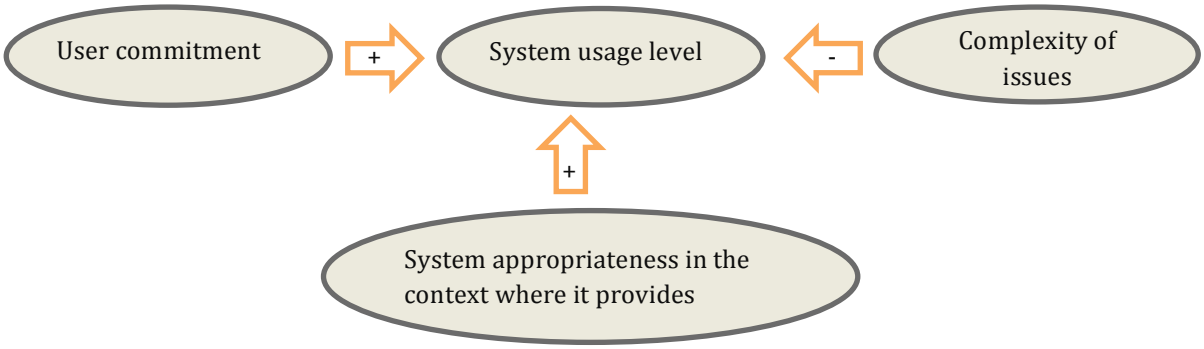

Figure 6 - The use of the system is strongly influenced by adaptation of the relationship between the system and the context where it must provide its support, the complexity of the issues, and the user commitment.

The mode of use of the system is an important success factor. The ***level of use of the system*** (factor 12) is positively influenced by the accuracy and relevance of the system output; it is the experience and knowledge of the decision-maker with respect to the domain. However, as already mentioned, the use depends on the complexity of the problems and the adequacy of the relationship between the system and the environment. ***Technology*** (factor 13) is a key variable for the effectiveness of a system. There is always the best tool, but you should choose the one that best fits the system context, not neglecting the constraint of limited resources. The choice of technology has a gap between availability on the market and the organization's ability to acquire and transform the technology effectively in a system (particularly if it is a decision support system). This ability is related both to the effort required to develop a system, that reflects the desired requirements of the system, and the commitment by management, convinced on

the help provided by the system in terms of improving decision-making and, therefore, on its strategic role.

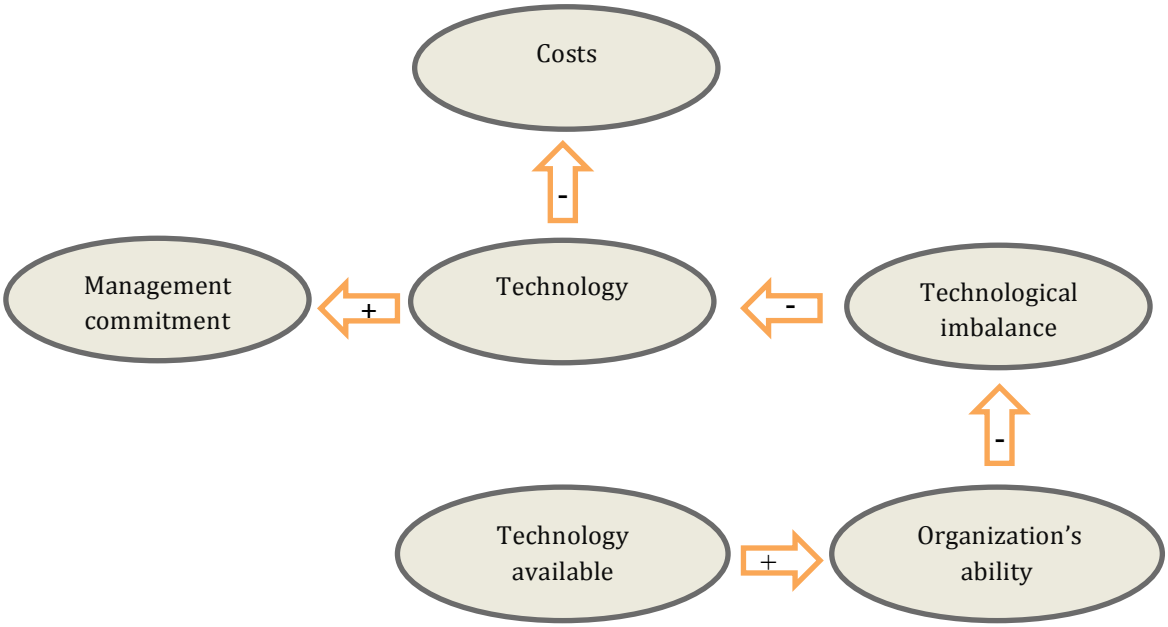

Figure 7 - Technology plays a fundamental role in determining the effectiveness of a system.

8.3.3 The quality of the system

Finally, as a last factor, we find the ***quality of the system*** (factor 14): it is influenced from usability, features and alignment between the system and the context for which it requests support. The usability is related to the quality due to the fact that the information for decision making may be provided in the required time. The functionality determines of a system is qualitatively valid, if is able to contribute to the solution of problems in the specific context.

This model can be put into practice by analyzing these 14 success factors and studying the causal relationships between the same variables using correlation coefficients. The results obtained would be compared with the theoretical model and one can validate the success or failure of the system. The correlation coefficients facilitate the understanding of the factors that significantly influence the system, and take action where it is considered appropriate to increase the quality of the system.

Despite the guidelines for proper decision-making process, the organizations naturally commit mistakes. When the uncertainty about the identification of the problem is quite high, the solution are not easily found, and it is very difficult to predict the outcome of the decision, errors are inevitable. Sometimes it happens that an error can be a source of success; on the one hand the errors are useful to acquire new information for the improvement of the planned solution; on the

other hand, strengthening the ability of decision makers increase their useful experience for future decisions. However, it is important to understand when an error leading to an interruption of the action without fear of the consequences.

The study of decision-making processes, the ability to analyze and decompose mechanisms, and especially the development of methodological and technical tools for support, is essential to achieve 'good' decisions. It is often the same decision-making process that produces significant results beyond the decisions and actions to which it leads; and this is due to its characteristic of being a learning process that somehow changes the actors themselves involved in it.

8.4 Decision-making process factors in Software Quality Assurance (SQA)

Based on the characteristics of the decision processes described in the previous paragraphs, and based on experience of the companies involved in the ICEBERG project , 7 main factors have been identified:

- **Actual Quality Level**
- **Expected Quality Level**
- **Human Factors**
- **Economics**
- **Time**
- **Resources**
- **Process**

Each of these factors should be taken into account to a properly decision related to Quality Assurance (and not only). It's very important to collect as many information as possible, related to the process and to the product, in order to take correct decisions.

The ICEBERG project partners are conducting a survey whose aim is to gather feedback on how companies operate on the factors related to the quality of the software.

8.4.1 Actual Quality Level

Actual Quality Level (AQL), represents the level of quality measured in a specific product / service before starting decision-making process. This is an input for decision-making process and can't be changed because it's a fixed measure (it refers to the past).

For a new project, it may be derived from the state of similar projects available in the context in which we work.

This factor can be determined measuring a series of sub-factors related to project historical quality metrics (as suggested by ISO/IEC 9126 standard):

External Quality Metrics: Metrics that can be collected observing running software. Example of external metrics are *functional test bug, integration test bug, result of performance test*

Internal Quality Metrics: Metrics that can be collected observing software internally. Example of internal metrics are: *code vomplexity, %comments, %duplications, rules compliance, maintainability level*

In use Metrics: Quality in use metrics are only available when the final product is used in real conditions. It is the effect of the using the software, rather than the quality of software itself. In other words, it tells how the system supports the user to get their job done, or how effective is the system usage. Example of in use metrics are: *production incident, production problem, production bug*.

8.4.2 Expected Quality Level

Expected Quality Level (EQL) is what the customer expects from a product or a service. We can also say it being a number of needs that are expected to be met.

It's possible define 2 needs type:

- **Explicit** needs, that will become specific;
- **Implicit** needs, what the customer does not tell anyone but he expects to be satisfied, however.

During a decision making process it is very important to have clear the EQL of the concerned object (product or service).

To do this, it is possible to get ideas from ISO standard for defining factors involved. Many factors are involved as suggest by ISO/IEC 25000 (ex ISO/IEC 9126) model:

Functionality: a set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs;

Reliability: a set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time;

Usability: a set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users;

Efficiency: a set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions;

Maintainability: a set of attributes that bear on the effort needed to make specified modifications;

Portability: a set of attributes that bear on the ability of software to be transferred from one environment to another.

8.4.3 Human Factors

In all decision-making processes, human factor is considered a fundamental characteristic.

The Software Quality Assurance is intended to provide a guarantee that a product / service complies with all the requirements. There is a trust aspect that is taken into account by Software Quality Assurance decision-maker.

There are many factors involved in this characteristic, many of which can require collecting information about people.

Some of these are:

Competence: level of competence on tools, technologies, domain, methodologies are all factors concerned to Competence. These are normally taken into account in all decision making process.

Experience: having people in the project that have already experience on similar projects may be an important thing.

Team spirit: flexibility, stable allocation, presence of a leader, cultural differences, ways of interaction, turnover of resources involved in project are factors linked to the team spirit.

8.4.4 Economics

This is a factor usually considered important in all the companies in the initial phase of each project. The cost of project can be estimated with different techniques. This phase is very important because it defines the budget for each activity, among which the Software Quality Assurance.

8.4.5 Time

The project *milestones* are always present. The Time factor must be taken into account in order to ensure that the project is "on time". Often, this is a problem for Quality Assurance activities, because a possible delay in development phase can compress the QA time.

8.4.6 Resources

During the project lifecycle, many resources are involved. It is possible to identify many type of resources:

- *Physical resources*: offices, desks, servers, phones;
- *Software resources*: tools to facilitate measuring and collection of information about product/service.

Regarding the Software Quality Assurance, there are many tools that support this phase:

- *Tools for test design and management:* these tools help you to create test cases, or at least test inputs (which is part of a test case);
- *Tools for bug recording (bug tracker):* these tools are designed to keep track of reported software bugs in software development efforts.
- *Tools for static code analysis:* Static analysis is the analysis of a software program that is performed without actually executing it. In most cases the analysis is performed on some version of the source code and in the other cases on some form of the object code. The term is usually applied to the analysis performed by an automated tool.
- *Tools for unit test execution:* unit testing is a method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures are tested to determine if they are fit for use. Tools for unit test support these activities.
- *Tools for automated test execution:* test automation is the use of special software (separate from the software being tested) to control the execution of tests and the comparison of actual outcomes to the predicted ones. Test automation can avoid manual execution of some repetitive, but necessary, tasks in a formalized testing process already in place, or add further test cases that would be difficult to perform manually.
- *Tools for performance / loading test:* are tools that in general test how a system performs in terms of responsiveness and stability under a particular workload.

8.4.7 Process

In IT, the **software process** term indicates the number of steps or paths to be carried out to obtain high quality results in a fixed time, working on the development of a product or system software.

It cannot be identified a quality process separated from the development; the two activities are closely related to each other, and are connected with the Project Management Process

Usually, it is possible to identify the following phases inside the **Project Management Process** (PMBOK approach):

- **Initiation**: Project technical proposal and estimation;
- **Plan**: Project planning phase;
- **Execution**: Project plan execution;
- **Controlling**: Project assessment and control;
- **Closing**: Project closure.

All the mentioned factors, referring to the decision making process in general, and then specifically referring to quality assurance systems, need to be included in the loop of a proper decision-maker design. The ICEBERG project partners will give a greater focus to the factors related to the quality assurance activities, presented in the last paragraph. To go more in depth, and figure out how companies deal with the outlined needs, the ICEBERG project partners are conducting a survey whose aim is to gather feedback on how companies operate

on the factors related to the quality of the software. This will serve as further basis for the successive phase of model-based process definition to support decision making about products quality and its associated cost.

9 REFERENCES

A. L. Goel, K. (1979). Time-dependent error-detection rate model for software reliability and other performance measures. *28* (3), 206-211.

A.Avritzer, & E.J.Weyuker. (2004). The role of modeling in the performance testing of e-commerce applications. *IEEE Transactions Software Engeneering , 30* (12), 1072–1083,.

Abdel-Hamid, T. K., & Madnick, S. (1991). *Software project dynamics. An integrated approach.* New Jersey: Prentice-Hall.

Abreu, F. B., & Carapuca, R. (1994). Object-oriented software engineering: Measuring and controlling the development process. *Fourth international conference on software quality.*

Acuña, S., Gómez, M., & Juristo, N. (2008). Towards understanding the relationship between team climate and software quality--a quasi-experimental study. *Empirical Software Engineering, 13(4)* , 401-434.

Adrion, W. R., Branstad, M. A., & Cherniavsky, J. C. (1982). Verification, Validation, and testing of Computer Software. *ACM Computing Surveys, 14 (2)* , 159-192.

Aggarwal, K. K., Singh, Y., Chandra, P., & Puri, M. (2005). Measurement of Software Maintainability Using a Fuzzy Model. *Journal of Computer Sciences , 1* (4), 538-542.

Aggarwal, K. K., Singh, Y., Kaur, A., & Malhotra, R. (2006). Application of Artificial Neural Network for Predicting Maintainability using Object -Oriented Metrics,. *Transactions on Engineering, Computing and Technology , 15*, 285-289.

Ahn, Y., Suh, J., Kim, S., & Kim, H. (2003). The Software Maintenance Project Effort Estimation Model Based on Function Points. *15* (2), 71-85.

AL-Ghamdi, A. S.-M. (2013). A Survey on Software Security Testing Techniques. *International Journal of Computer Science and Telecommunications , 4* (4), 14-18.

APM. (2010). *A History of the Association for Project Management 1972-2010.* Buckinghamshire: Association for Project Management.

Avizienis, A., Laprie, J.-C., Randell, B., & Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on , 1* (1), 11-33.

Bach, J. (1995). Enough About Process: What We Need Are Heroes. *IEEE Computer, 12(2)* , 96-98.

Bach, J. (1999). What Software Reality Is Really About. *IEEE Computer, 32(12)* , 148-149.

Bandini, S., Paoli, F. D., Manzoni, S., & Mereghetti, P. (2002). A support system to COTS based software development for business service. *Proceedings of the 14th*

International Conference on Software Engineering and Know ledge Engineering, (pp. 307-314).

Bansiya, .., & Davis, C. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering, , 28* (1), 4-17.

Basili, V. R., Briand, L. C., & Melo, a. W. (1996). A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering , 22* (10), 751-761.

Basili, V., & Reiter R. W., J. (1979). An Investigation of Human Factors in Software Devel-opment. *IEEE Computer, 12(12)* , 21-38.

Beaver, J., & Schiavone, G. A. (2006). The effects of development team skill on software product quality. *SIGSOFT Software Engineering Notes, 31 (3)* , 1-5.

Becker, S., Grunske, L., Mirandola, R., & Overhage, S. (2006). Performance prediction of component-based systems: A survey from an engineering perspective. *RCHITECTING SYSTEMS WITH TRUSTWORTHY COMPONENTS, VOLUME 3938 OF LNCS* , 169-192.

Beecham, S., Baddoo, N., Hall, T., Robinson, H., & Sharp, H. (2008). Motivation in Software Engineering: A systematic literature review. *Information and Software Technology, 50* , 860-878.

Binkley, A., & Schach, S. (1998). Validation of the coupling de- pendency metric as a predictor of run-time failures and maintenance measures. *Proceedings of the Intl. Conference on Software Engineering* (pp. 452-455). IEEE.

Bobbio, A., & Trivedi, K. (1986). An Aggregation Technique for the Transient Analysis of Stiff Markov Chains. *IEEE Transactions on Computers , C-35* (9), 803-814.

Boehm, B. W. (1981). *Software engineering economics.* New Jersey: Prentice-Hall.

Brooks, F. (1975). *The mythical man-month.* Boston: Addison-Wesley.

Caglayan, B., Tosun, A., Miranskyy, A., Bener, A., & Ruffolo, a. N. (2010). Usage of multiple prediction models based on defect categories. *Proceedings of the International Conference on Predictive Models in Software engineering.*

Cai, K. Y., Gu, B., Hu, H., & Li, a. Y. (2007). Adaptive software testing with fixed-memory feedback. *Journal of Software and Systems , 80* (8), 1328-1348.

Cai, Y. K., Li, C., & Liu, K. (2004). Optimal and adaptive testing for software reliability assessment. *Information Software Technology , 46* (15), 989-1000.

Carrozza, G., Cotroneo, D., Natella, R., Pietrantuono, R., and Russo, S. (2013). Analysis and Prediction of Mandelbugs in an Industrial Software System . *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, (pp. 262-271).

Catal, C., & Diri, B. (2009). A systematic review of software fault prediction studies. *Expert Systems with Applications , 36* (4).

CENELC 50126. (1999). *Railway applications – The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS) – Part 1: Basic requirements and generic process.*

CENELEC 50128. (2011). *Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems.*

CENELEC 50129. (2003). *Railway applications – Communication, signalling and processing systems – Safety related electronic systems for signalling.*

Chen, T. Y., Kuo, F.-C., & and H. Liu. (2009). Application of a failure driven test profile in random testing,. *IEEE Transactions on Reliability , 58* (1), 179-192.

Cheney, L. H. (1984). Effects of individual characteristics, organizational factors and task characteristics on computer programmer productivity and job satisfaction. *Information Management, 7(4)* , 209-214.

Chidamber, S., & Kemerer, C. (1994). AMetricsSuiteforOb- ject Oriented Design. *IEEE Transactions on Software En- gineering , 20* (6).

Chou, A., Yang, J., Chelf, B., Hallem, S., & Engler, D. (2001). An empirical study of operating systems errors. *Proc. ACM Symp. on Operating Systems Principles.*

Ciardo, G., & Trived, K. S. (1993). Decomposition Approach to Stochastic Reward Net Models. *Performance Evaluation , 18* (1), 37-59.

CMMI Product Team. (2006). *CMMI for Development, Version 1.2,CMMI-DEV, V1.2, CMU/SEI-2006-TR-008, ESC-TR-2006-008.* Pittsburgh : Software Engineering Institute.

Cohn, M. (2009). *Succeeding with Agile: Software Development Using Scrum.* Reading: Addison-Wesley.

Cotroneo, D., Natella, R., and Pietrantuono, R. (2013). Predicting aging-related bugs using software complexity metrics. *Performance Evaluation , 70* (3), 163-178.

Cotroneo, D., Pietrantuono, R., and Russo, S. (2013). Testing techniques selection based on ODC fault types and software metrics. *Journal of Software and Systems , 86* (6), 1613-1637.

Couger, J. D., & Zawacki, R. A. (1980). *Motivating and managing computer personnel .* New York: Wiley lnter-science.

D. G., A. P., & Emmerich, W. (2004). Early performance testing of dis- tributed software applications. *Proceedings of the 4th International Workshop on Software and Performance.*

D.Long, A.Muir, & R.Golding. A Longitudinal Survey of Internet Host Reliability. *Proceedings of the 14th Symposium on Reliable Distributed Systems.*

Dai, Y.-S., Pan, Y., & Zou, X. (2007). A Hierarchical Modeling and Analysis for Grid Service Reliability. *IEEE Transactions on Computers , 56* (5), 681-691.

Denaro, G., & Pezze, M. (2002). An Empirical Evaluation of Fault-proneness Models. *Proceedings 24th Intl. Conf. on Software Engineering,* (pp. 241-251). IEEE.

Denaro, G., Morasca, S., & Pezze, M. (2002). Deriving Models of Software Fault-proneness. *Proceedings 14th International Conference on Software Engineering and Knowl- edge Engineering* (pp. 361-368). IEEE.

Duraes, J., & Madeira, H. (2002). Characterization of Operating Systems Behavior in the Presence of Faulty Drivers through Software Fault Emulation. *Proc. Pacific Rim Intl. Symp. on Dependable Computing* , 201-209.

Fenton, N. E., & Pfleeger, S. L. (1998). *Software Metrics: A Rigorous and Practical Approach (2nd ed.).* Boston, MA, USA.: PWS Pub. Co.

Fernández, L. (2009). Personal Skills for Computing Professionals. *IEEE Computer, 42(10)* , 110-112.

Fernández, L., & García, M. (2003). Software engineering professionalism. *Upgrade, 4* , 42-46.

Fernández, L., & Misra, S. (2012). Analysis of cultural and gender influences on teamwork performance for software requirements analysis in multinational environments. *IET Software, 6(3)* , 167-175.

Ferrari, A. (2010). *Fattori critici per il successo dei progetti*. From ZeroUno: http://www.zerounoweb.it/approfondimenti/business-intelligence/fattori_critici_per_succebo_dei_progetti.html

Fioravanti, F., & Nesi, P. (2001). Estimation and Prediction Metrics for Adaptive Maintenance Effort of Object - Oriented Systems. *IEEE Transactions on Software Engineering , 27* (12), 1062-1084.

Fleckenstein, W. O. (1983). Challenges in software development. *IEEE Computer, 16(3)* , 60-64.

Folmer, E., & Bosch, J. (2004). Architecting for usability: a survey. *The Journal of Systems and Software , 70*, 61-78.

Ganesh, J. P., & Dugan, J. B. (2002). Automatic Synthesis of Dynamic Fault Trees from UML System Models. *Proc. of the IEEE Int. Symposium on Software Reliability Engineering*, (pp. 243-256).

Garzia, M. (2002). Assessing the Reliability of Windows Servers. *Proc. of Dependable Systems and Networks.* IEEE.

Gill, N. (2005). Factors affecting effective software quality management revisited. *SIGSOFT Software Engineering Notes, vol. 30, 2* , 1-4.

Goel., A. L. (1985). Software Reliability Models: Assumptions, Limitations and Applicability. *IEEE Trans. on Software Engineering , SE-11* (12), 1411-1423.

Gokhale, S., & Lyu, M. (1997). Regression Tree Modeling for the Prediction of Software Quality. *Proceedings of the 3rd ISSAT .*

Gokhale, S., & Trivedi, K. (1998). Log-logistic software reliability growth model. *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium*, (pp. 34-41).

Gokhale, S., E.Wong, W., Horganc, J., & Trivedi, K. S. (2004). An analytical approach to architecture- based software performance and reliability prediction. *Performance Evaluation , 58* (4), 391-412.

Gokhale, S., Lyu, M., & Trivedi, K. (2006). Incorporating fault debugging activities into software reliability models: A simulation approach. *IEEE Transactions on Reliability , 55* (2), 281-292.

Goldenson, D. R., & Gibson, D. L. (2003). *Demonstrating the Impact and Benefits of CMMI: An Update and Preliminary Results, CMU/SEI-2003-SR-009.* Pittsburgh: Software Engineering Institute.

Goseva-Popstojanova, & Trivedi. (2001). Architecture-based approach to reliability assessment of software systems. *Performance Evaluation , 45* (2-3), 179-204.

Goseva-Popstojanova, K., Mathur, A., & Trivedi, K. (2001). Comparison of architecture-based software reliability models. *Proc. of the 12th International Symposium on Software Reliability Engineering (ISSRE '01)*, (pp. 22-31).

Griesser, J. W. (1993). Motivation and information system professionals. *Journal of Managerial Psychology, 8(3)* , 21-30.

Guterl, F. (1984). Spectrum/Harris poll - The job. *IEEE Spectrum, 21(6)* , 38.

Haberkorn, M., & Trivedi, K. (2007). Availability Monitor for a Software Based System. *Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium*, (pp. 21-238).

Halstead, M. (1977). *Elements of Software Science.* Elsevier Science.

Harrington, H. J. (1987). *Poor-quality cost.* New York: McGraw-Hill.

Herbsleb, J. D., Carleton, A., Rozum, J. A., Siegel, J., & Zubrow, D. (1994). *Benefits of CMM-Based Software Process Improvement: Initial Results.* Pittsburgh: SEI.

Hoegl, M., & Gemuenden, H. (2001). Teamwork Quality and the Success of Innovative Projects: A Theoretical Concept and Empirical Evidence. *Organization Science, 12(4)* , 435-44.

Hofstede, G. (2001). *Culture's Consequences: comparing values, behaviours, institutions, and organizations across nations.* SAGE Publication.

Hsia, P., Hsu, C.-T., & Kung, D. (1999). Brooks' Law Revisited: A System Dynamics Approach. . *In 23rd International Computer Software and Applications Conference (COMPSAC'99)* (pp. 370-375). Washington, DC, USA: IEEE Computer Society.

Hsueh, M.-C., Tsai, T., & Iyer, a. R. (1997). Fault injection techniques and tools. *Computer , 30* (4), 75-82.

IEEE 1012. (2012). *1012-2012 - IEEE Standard for System and Software Verification and Validation.* IEEE.

IEEE 1044-2009. (2009). *IEEE Standard Classification for. Software Anomalies.* IEEE Computer Society.

IEEE 610.12. (n.d.). IEEE Standard Glossary of Software Engineering Terminology.

IEEE 729-1993. (1993). IEEE Software Engineering Standard 729-1993: Glossary of Software Engineering Terminology. IEEE Computer Society Press.

IEEE. (1984). *IEEE Std. 730, Standard for Software Quality Assurance Plans.* New York: IEEE.

IEEE Std 1003.1b. (1993). *IEEE Standard for Information Technology-Portable Operating System Interface (POSIX).*

ISO 15504-1. (2004). *Information technology-Process assessment -Part 1: Concepts and vocabulary.* Geneve: ISO.

ISO 26262. (2011). *Road vehicles-Functional safety.*

ISO 9001. (2008). *Quality management systems -Requirements.* Geneve: ISO.

ISO/IEC 12207-2008. (2008). *Systems and software engineering — Software life cycle processes.* IEEE Computer Society.

ISO/IEC 15504. (2012). *15504:2012 - Information technology — Process assessment.*

ISO/IEC 29119. (2013). *ISO/IEC 29119 Software Testing- The international software testing standard.* ISO/IEC.

ISO/IEC 90003. (2004). *Software engineering - Guidelines for the application of ISO 9001:2000 to computer software.* Geneve: ISO.

ISO/IEC 9126. (2001). *Software engineering-Product quality.*

J. F. Meyer, & W.H.Sander. (1993). Specification and Construction of Performability Models. *Int. Workshop on Performability Modeling of Computer and Communication Systems*, (pp. 1-32).

Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The Unified Software Development Process.* Reading: Addison-Wesley.

Jiang, Y., Cukic, B., & Menzies, T. (2007). Fault prediction using early lifecycle data. *Eighteenth IEEE international symposium on software reliability*, (pp. 237-246).

Jing, X. (2010). A Brief Survey on the Security Model of Cloud Computing. *Distributed Computing and Applications to Business Engineering and Science (DCABES), 2010 Ninth International Symposium on*, (pp. 475-478).

Johansson, A., Suri, N., & Murphy, a. B. (2007). On the selection of error model(s) for OS robustness evaluation. *Proc. Intl. Conf. on Dependable Systems and Networks*, (pp. 502–511).

Johansson, A., Suri, N., & Murphy, B. (2007). On the impact of injection triggers for OS robustness evaluation. *ISSRE.*

Jones, C. (1994). *Assessment and control of software risks.* Saddle River: Yourdon Press.

Jones, C. (1998). *Estimating software costs.* Hightstown, NJ: McGraw-Hill.

Jones, C. (1998). *Estimating software costs.* Hightstown, NJ: McGraw-Hill, Inc.

K.Y.Cai. (2002). Optimal software testing and adaptive software testing in the context of software cybernetics,. *Information Software Technology , 44* (14), 841-855.

Kanoun, C. B.-A. (2004). Construction and Stepwise Refinement of Dependability Models. *Performace Evaluation , 56*, 277-306.

Khairuddin, H., & Elizabeth, K. (1996). A Software Maintainability Attributes Model. *Malaysian Journal of Computer Science , 9* (2), 92-97.

Khoshgoftaar, T., Gao, K., & Szabo, R. (2001). An application of zero-inflated poisson regression for software fault prediction. *Twelfth international symposium on software reliability engineering,.* IEEE.

Koopman, P., & DeVale, J. (2002). The exception handling effectiveness of POSIX operating systems. *IEEE Trans. on Software Engineering , 26* (9).

Koopman, P., Sung, J., Dingman, C., Siewiorek, D., & Marz, T. (1997). Comparing operating systems using robustness benchmarks. *SRDS.*

Koziolek, H. (2010). Performance Evaluation of Component-based Software Systems: A Survey. *Performance Evaluation , 67* (8), 634– 658.

Krautsevich, L., Martinelli, F., & Yautsiukhin., A. (2010). Formal approach to security metrics: what does “more secure” mean for you? *Proceedings of 4th European Conference on Software Architecture: Companion Volume, ,* (pp. 162-169).

Laverdiere, M. A., Mourad, A., Hanna, A., & Debbabi, M. (2006). Security Design Patterns: Survey and Evaluation . *Electrical and Computer Engineering, 2006. CCECE '06. Canadian Conference on.*

Lientz, B. P., & Swanson, E. B. (2000). *Software Maintenance Management.* Addison - Wesley.

Liu, Y., Gorton, I., Liu, A., Jiang, N., & Chen, a. S. (2002). Designing a test suite for empirically-based middleware performance prediction,. *Proceed- ings of the Fortieth International Conference on Tools Pacific Objects for internet, mobile and embedded applications series.*

Lollini, P., Bondavalli, A., & Giandomenico, F. D. (2009). A decomposition-based modeling framework for complex systems. *IEEE Transactions on Reliability* .

Lorenz, M., & Kidd, J. (1994). *Object-oriented software metrics: A practical guide.* Prentice Hall, Inc.

Mısırlı, A., Caglayan, B., Miranskyy, A., Bener, A., & Ruffolo, a. N. (2011). Different strokes for different folks: a case study on software metrics for different defect categories. *Proceedings of teh internatlional workshop. on Emergerging Trends in Software Metrics* , (pp. 45-51).

McCabe, T. (1976). A complexity measure . *IEEE Transactions on Software Engineering , 2* (4), 308-320.

McConnel, S. (1996). *Rapid Development: Taming Wild Software Schedules.* Redmond: MS Press.

Menzies, T., Greenwald, J., & A.Frank. (2007). Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering , 33* (1), 2-13.

Mills, H. D., Dyer, M., & R. C. Linger. (1987). Cleanroom software engi- neering. *Software , 4* (5), 19-24.

Mishra, K., & Trivedi, K. (2006). Model Based Approach for Autonomic Availability Management. *Proc. of the Intl. Service Availability Symposium*, *4328*, pp. 1-16.

Misra, S., & Fernández-Sanz, L. (2011). Quality Issues in Global Software Development. *The Sixth International Conference on Software Engineering Advances (ICSEA'2011)*, (pp. 325-330). Barcelona.

Montanari, F. (2005). Le distorsioni cognitive nei processi decisionali e negoziali: una review e alcuni esperimenti. *ticonzero* .

Mullen, R. (1998). The lognormal distribution of software failure rates: application to software reliability growth modeling. *Proceedings 9th International Symposium on Software Reliability Engineering*, (pp. 134-142).

Murugesan, S. (1994). Attitude towards testing: a key contributor to software quality. *Proceedings of First International Conference on Software Testing, Reliability and Quality Assurance*, (pp. 111-115).

Musa, J. D. (1996). Software-reliability-engineered testing. *Computer , 29* (11), 61-68.

Nagappan, N., Ball, T., & Zeller, A. (2006). Mining Metrics to Predict Component Failures. *Proceedings of the 28th International conference on Software engineering* , pp. 452-461.

Nam, J., Jialin Pan, S., & and Kim, S. (2013). Transer Defect Learning. *Proceedings of the International conference on software engineering.*

NASA . (2009 йил 28-1). *Software Assurance Definitions*. From http://www.hq.nasa.gov/office/codeq/software/umbrella_defs.htm

Natella, R. (2011). *Achieving Representative Faultloads in Software Fault Injection.* PhD thesis, Universit`a degli Studi di Napoli Federico II,.

Nicol, D., Sanders, W., & Trivedi, K. (2004). Model-based evaluation: From dependability to security. *IEEE Transactions on Dependable and Secure Computing* , 48-65.

Nielsen, J. (1993). *Usability Engineering.* Academic press.

Ohlsson, N., & Alberg, H. (1996). Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering , 22* (12), 996-894.

Okamura, H., Dohi, T., & Osaki, a. S. (2004). Em algorithms for logistic software reliability models. *Proc. 22nd IASTED International Conference on Software Engineering*, (pp. 263-268).

Ostrand, T., Weyuker, E., & Bell, R. (2005). Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering , 31* (4), 340-355.

Ostrand, T., Weyuker, E., & Bell, R. (2005). Predicting the Location and Number ofFaults in Large Software Systems. *IEEE Transactions on Software Engineering , 31* (4), 340-355.

Pezze, M., & Young, M. (2008). *Software Testing and Analysis: Process, Principles and Techniques.* John Wiley & Sons.

Pietrantuono, R., Russo, S., and Trivedi, K. (2010). Software reliability and testing time allocation: An architecture-based approach. *IEEE Transactions on Software Engineering , 36* (3), 323-337.

Poulding, S., & Clark, J. A. (2010). Efficient software verification: Statistical testing using automated search. *IEEE Transactions Software Engineering , 36* (6), 763-777.

Pressman, R. S. (2009). *Pressman, R. S.: Software Engineering Software Engineering: A Practitioner's Approach. 7th Edition.* McGraw-Hill.

Pusatli, O. T., & Misra, S. (2011). A discussion on assuring software quality in small and medium software enterprises: an empirical investigation. *Tehnički vjesnik, 18(3)* , 447-452.

R.Pietrantuono, S. Russo, and K. Trivedi, (2010). Online monitoring of software system reliability. *Dependable Computing Conference (EDCC), 2010 European* (pp. 209-218). Valencia: IEEE.

Rabah, M., & K. Kanoun. (2003). Performability evaluation of multipurpose multiprocessor systems: the "separation of concerns" approach. *IEEE transactions on Computers , 52* (2), 223-236.

Ren-zuo, X., Ruo-feng, M., L., L.-n., & Zhong-wei, X. (2004). Human Factors Analysis in Soft-ware Engineering. *Wuhan University Journal of Natural Sciences, 9(1)* , 18-22.

Rico, D. (2004). *Rico, D.F.: ROI of Software Process Improvement: Metrics for Project Managers and Soft-ware Engineers.* J. Ross Publishing.

Roy, A., Kim, D. S., & Trivedi, a. K. (2012). Attack countermeasure trees (ACT): towards unifying the constructs of attack and defense trees. *Security and Communication Network , 5* (8), 929-943.

RTCA - DO 178C. (2011). *Software Considerations in Airborne Systems and Equipment Certification.*

Rumsfeld, D. (2002). There are known knowns.

Sarbu, C., Johansson, A., Suri, N., & Nagappan, N. (2009). Profiling the operational behavior of OS device drivers. *Empirical Software Engineering , 15* (4).

Schulmeyer, G. (1992). The net negative producing programmer. *American Programmer, 6* .

Seliya, N., Khoshgoftaar, T., & Hulse, J. V. (2010). Predicting Faults in HighAssurance Software. *Proceedings IEEE 12th Intl. Symp. on High AssuranceSystems Engineering*, (pp. 26-34).

Shackel, B. (1991). Usability–context, framework, design and evaluation. *In: Shackel, B., Richardson, S. (Eds.), Human Factors for Informatics Usability.* , 21-38.

Shukla, R., & Mishra, A. K. (2008). Estimating Software Maintenance Effort - A Neural Network Approach. *Proceedings of the 1st conference on India Software Engineering Conference*, (pp. 107-112).

Silva, G., & Madeira, H. (2005). Experimental dependability evaluation. *In Diab, H.B., Zomaya, A.Y., eds.: Dependable Computing Systems: Paradigms, Performance Issues, and Applications* (pp. 319-347). Wiley.

Simon, H. (1960). *The New Science of Management Decision.* New York: Harper & Row.

Singh, Y., Kaur, A., & Sangwan, O. P. (2004). Neural Model for Software Maintainability. *Proceedings of International Conference on ICT in Education and Development*, (pp. 1-11).

Software Engineering Institute. (2010). *Capability Maturity Model Integration , verion 1.3.* SEI.

Sommerville, I. (2004). *Software Engineering.* Addison-Wesley,.

Subramanyam, R., & Krishnan, M. S. (2003). Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. *IEEE Transactions on Software Engineering , 29* (4), 297-310.

T.Y.Chen, F.-C.Kuo, & Liu, H. (2008). Distributing test cases more evenly in adaptive random testing. *Journal of Software and Systems , 81* (12), 2146-2162.

Tang, D., & Iyer, R. (1993). Dependability Measurement and Modeling of a Multicomputer System. *IEEE Transactions on Computers , 42* (1), 62-75.

Thomas, S. A., Hurley, S. F., & J.:, B. D. (1996). Looking for the human factors in software quality management . *Proceedings of the 1996 International Conference on Software En-gineering: Education and Practice (SEEP '96)* (pp. 474-480). Washington: IEEE Computer Society.

Ting-Peng, L., Jiang, J., Klein, G., & Liu, J.-C. (2010). Software Quality as Influenced by Informational Diversity, Task Conflict, and Learning in Project Teams. *IEEE Transactions on Engineering Management, 57(3)* , 477-487.

Tiwari, G., & Sharma, A. (2012). Maintainability Techniques for Software Development Approaches – A Systematic Survey. *Special Issue of International Journal of Computer Applications on Issues and Challenges in Networking, Intelligence and Computing Technologies* .

Vaidyanathan, K., & Trivedi, K. S. (2005). A comprehensive model for Software Rejuvenation. *IEEE Transactions on Dependable and Secure Computing , 2* (2), 124-137.

van Solingen, R. (2006). Calculating Software Process Improvement's Return on Investment. *Advances in Computers, 66* , 1-41.

W.Wang, Y.Wu, & Chen, M. (2009). An architecture-based software reliability model. *Proc. of the Pacific Rim Dependability Symposium.*

Weyuker, E., & Vokolos, F. (2000). Experience with performance testing of software systems: issues, an approach, and case study. *Software Engineering, IEEE Transactions on , 26* (12), 1147–1156.

Wong, B., & Bhatti, M. (2009). The influence of team relationships on software quality. *ICSE Workshop on Software Quality WOSQ'09* , 1-8.

Woodruff, C. K. (1980). Data processing people - Are they really different? *Information & Management, 3(4)* , 133-139.

Y. Liu, A. F. (2005). Design level performance prediction of component-based applications. *IEEE Transactions on Software Engineering , 31* (11), 928–941.

Y., L., A., F., & I., G. (2005). Design-level performance prediction of component-based applications,. *IEEE Transactions on Software Engineering , 31* (11), 928–941.

Yamada, S., Ohba, M., & Osaki, a. S. (1983). S-Shaped Reliability Growth Modeling for Software Error Detection. *IEEE Trans. on Reliability , R-32* (5), 475-485.

Zhang, Z. (2001). *Overview of usability evaluation methods.* From <http://www.cs.umd.edu/~zzj/UsabilityHome.html>.

Zhou, Y., & Leung, H. (2006). Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Transactions on Software Engineering , 32* (10), 771-789.

Zimmermann, T., Nagappan, N., Gall, H., Giger, E., & Murphy, B. (2009). Cross-project Defect Prediction-A Large Scale Experiment on Data vs. Do-main vs. Process. *Proceedings 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Sym- posium on theFoundations of Software Engineering*, (pp. 91-100).

Zuser, W., & Grechenig, T. (2003). Reflecting Skills and Personality Internally as Means for Team Performance Improvement. *In Proceedings of the 16th Conference on Software Engineering Education and Training (CSEET '03).* Washington: IEEE Computer Society.