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.:	***2.2***
Deliverable Title:	*Validation scenarios and quality parameters*
Organisation name of lead Contractor for this Deliverable:	*ANET, DEISER*
Author(s):	*L. Fernandez, P. Potena, N. Condori, L. de Marcos, M. Yatskevich, D. Rodriguez, I. Battipaglia, C. Gaiani, Guillermo Montoya*
Participant(s)	*All*
Work package contributing to the deliverable:	*2*
Task contributing to the deliverable:	*T 2.4, T2.5*
Total Number of Pages	*83*

Table of Versions

Version	Date	Version Description	Contributors
0.1	06-02-2014	Draft. First document skeleton.	Luis Fernandez Pasqualina Potena Nelly Condori Luis de Marcos
0.2	07-04-2014	Added Criteria, Metrics and Evaluation Scenarios sections.	Mikalai Yatskevich
0.3	10-04-2014	Updated document structure and Evaluation Scenarios section.	Claudio Gaiani
0.4	30-04-2014	Updated document. Added existing categories of applicable decision models and Summary of indicators for decision models sections.	Luis Fernandez Pasqualina Potena Luis de Marcos
1.0	30-04-2014	Updated Evaluation Scenarios section.	Mikalai Yatskevich Luis de Marcos Claudio Gaiani
1.1	23-05-2014	Updated Evaluation Scenarios Section	Guillermo Montoya
1.2	06-11-2014	Updated Evaluation Scenario Section	Mikalai Yatskevich

TABLE OF CONTENTS

1 EXECUTIVE SUMMARY

The aim of D2.2 of the ICEBERG project "Validation scenarios and quality parameters" is to identify and classify parameters of interest for the definition of the models. The secondary aim is to select the metrics and describe the quality attributes of interest (e.g., number of defects found, operational reliability, robustness). The tertiary aim is to obtain a set of scenarios for suitably validating the implemented process. The test cases are selected based on a) relevancy of the topic; b) adaptability; c) scalability; d) extendibility. The document provides the ICEBERG partners with the guidelines to pursue the mentioned project aims.

2 INTRODUCTION

The definition of software quality is typically given by considering two levels: 1) the intrinsic product quality and 2) the customer related. Existing approaches for intrinsic product quality can be classified in i) defect management approaches, and ii) quality attributes approaches. The first ones are focused on defects management, whereas the second ones assess the software quality by considering quality factors (e.g., reliability, usability, interoperability, etc.). However, reality of software development professional practice does not enable choosing any of the above approaches, as they involve costs, prerequisites for implementation, etc.

The ICEBERG project aims at defining how project management decisions on quality assurance actions influence project's results in terms of intrinsic product quality while evaluating their effects in costs and schedule, following the idea of the Iron Triangle for project management. The reason is that software quality cannot be analyzed separately, because the project managers must assure the respect to constraints on schedule and costs. A quality decision, for example, can be the one of implementing static code analysis (e.g. tools, new processes, training, etc.) but its impact on project schedule, for example, can cause delays in completion of projects tasks while number of defects might be reduced up to certain extent leading to cost savings: in the end, the project manager need to know if this is helpful and convenient for the project goals.

Therefore, finding new ways of supporting the quality decision making process would make possible to understand in advance the real impact of decisions on cost (and schedule), and decide the corrective actions to be implemented at any level or category of decision (tools, QA or development processes, organizational factors, professional competences involved in the entire software development cycle, etc.). This analysis might be used to answer questions such as: (i) Given a high quality constraint, what is the cost to achieve, measurably, the goal? Is there a way to minimize such cost, standing the required high quality? (ii) Given a budget constraint (which prevents from performing all the required quality activities) what is the cost for the missing quality activities? Missing activities may imply bad quality: how does this “bad quality” manifests itself during operation, and how much does it cost?

This document is structured as follows. Section 3 describes the approach to address the goal of the ICEBERG project. Section 4 addresses measurement of the basic factors (cost and time). Section 5 describes the approach to defect management. Section 6 discusses software size. Section 7 describes how we intend to assess the measures for the three factors of the iron triangle. Section 8 provides an overview on exiting categories of decision models. Section 9 summarizes indicators for decisions models. Section 10 describes the validation scenarios that are used in the ICEBERG project. Finally, Section 11 concludes the deliverable.

3 HOW TO TACKLE WITH THE GOAL OF THE ICEBERG PROJECT

The Iceberg project aims at defining how project management decisions on quality assurance actions influence project's results in terms of quality, cost and time. This analysis is inspired in two underlying concepts:

- Cost of Quality (CoQ): this concept claims measuring the cost linked to achieving quality results in projects or in general in any development/production activity. In short, it states that getting quality requires investing money before, although if this is done correctly the ROI (Return of Investment) is positive leading to the popular motto "Quality is free" created by Crosby [1]. More specifically for the Iceberg project, the well-known refinement of CoQ named as the Cost of Poor Quality (CoPQ, first popularized by an IBM expert, Harrington [2]) has been adopted. CoPQ is defined as costs which are generated as a result of producing defective outcomes, products or services. Harrington himself insisted in the fact that preventing and pursuing quality makes it profitable because you can save CoPQ (he mentions declarations of managers claiming that one dollar in evaluation saves nine in losses and one in prevention leading to saving up to 15 in losses for failures and also typical cases of 20:1 [3]). Others have reported case studies showing ratios of 1 to 10 in return for each money unit (dollar) invested in prevention.

- The Iron Triangle: a concept which summarizes the three axis which project managers are supervising and where they are intervening with their decisions. First defined by [3], this concept has been refined in subsequent years by transforming it into a square by adding the dimension Scope to the three traditional ones, i.e. Time/Schedule, Cost/Money and Quality/Defects. The concept goes beyond stating the dimensions of the project which are supervised but also the importance of considering the influences between them, e.g. if circumstances or requests from stakeholders leads to reduce schedule, effects on the other dimensions are predicted. Although the classical view is that certain relations are always configured in the same way, e.g. less money would lead to poorer quality, this is not an automatic rule as other factors could also intervene, e.g. increasing productivity while keeping quality level: Harrington highlighted this in [4] remarking that the relation between productivity and quality is not contradictory but complementary.

In the Iceberg project, both conceptual frames are the inspiration for the goal pursued by the project:

- Defining relationships between project management decisions and effects on quality while observing associated consequences in costs and schedule, especially analyzing the trade-off between the diminishing of CoPQ with the investment in quality (sometimes understood as CoQ).

There is not a unique way of defining and measuring CoQ and CoPQ. Several works have analyzed systematically the proposals located in the literature (e.g. [5]). In general, at least these aspects of costs are evaluated: appraisal, prevention, internal failure and external failure costs. Experiments have confirmed several relationships in specific companies (e.g. wholesale in [5]):

- Inverse between appraisal and prevention costs and failure costs for company, materials and labour in different degrees (no clear relation for machines).
- Direct influence between appraisal and prevention costs and quality level for all factors.
- Quality enhances as a result of reduction of failures.

Moreover, many costs of CoQ are hidden and rarely captured by conventional accounting procedures and sometimes even considered as a regular cost of doing business. Maybe the most important ones are customer incurred, loss of reputation and customer dissatisfaction which impact on future purchasing decisions. Thus, eliminating external failures leads to elimination of most part of these costs (coherent with the third conclusion above) so they are having the highest priority.

However, the models for CoQ and CoPQ started as part of the general research streamline of industrial quality which provoke the rise of all models and concepts related to general quality management. This means that they were initially focused on the traditional settings of all these research efforts: industrial manufacturing organizations. Now the quality management discipline has been extended to all types of products and services although the most mature results still belong to the manufacturing sector.

In the case of software engineering, unlike other productive activities, the development of products (software, applications, etc.) is quite different to the vast majority of other production cases due to many reasons (e.g. see [7]): intellectual nature of products, no raw materials, no physical laws governing their behavior, main cost allocated to developing first copy not to creating copies, immaturity of market, repairing defects does not mean reverting to original state as just deliver from development, flexibility, etc. The consequence is that software quality management cannot just copy and adapt the solid and well-known techniques developed by manufacturing organizations or by other similarly mature productive activities. Software engineering has had to develop its own particular methods.

In the case of software quality, researchers have followed different approaches and have addressed varied scopes from industry and company to project level. A good systematic review of literature can be found in [8]. The article deals with all types of contributions in impact publications which mention specifically software quality costs: it excludes those related to software developments costs which do not clearly detail quality aspect and those related to SQA which do not analyze costs. A high number of articles were not empirically validating their proposals or models.

A whole line of work was proposed by B. Boehm [8] and named as Value-Based Software Engineering (VBSE) with the goal of enriching traditional techniques of software project management (e.g. Earned Value) by adding an evaluation of benefits realized by stakeholders. VBSE integrates value considerations into current and emerging software engineering principles and practices. Consideration of value provided by software is based on methods like the BRA (Benefits Realization Approach) [9]. Methods like BRA have certain prerequisites: accountability related to ownership, relevant measurement and proactive change.

Research has proven that humans make trade-off analyses continuously, and especially when deciding, if not on the ground of objective measurements then relying in intuition. As SQA is an investment with significant cost and sometimes with lack of quantification of benefits, clear evaluation and consideration of costs and benefits have to be provided.

4 COST AND TIME MEASUREMENT

One of the main shortcomings in creating an environment for evaluating decisions in QA is the traditional lack of consistent and organized measurement procedures in many companies. Taking into consideration the basic measurement of the three main effects which everybody wants to control in software projects (quality, cost and time), we can be shocked by the challenges we have to face. Starting with schedule and time, Capers Jones [11] highlights several limiting facts:

- Schedule data is also very troublesome and ambiguous. It is surprising its scarce presence in solid studies for decades and even more surprising that so few companies track software development schedules, although this topic is the most important to software managers and executives.
- At least 85 percent of the software managers in the world jump into projects with hardly a clue as to how long they will take. When collecting schedule information on historical projects, establishing the true schedule duration of a software project is a tricky task: when software is delivered is often fairly clear, but when it originated is imprecise data point.
- More than 15 percent of software projects are also ambiguous in determining when they were truly delivered. Sources of ambiguity are whether to count the start of external beta testing as the delivery point or wait until the formal delivery when beta testing is over. Another source of ambiguity is whether to count the initial delivery of a software product, or whether to wait until deferred functions are completed and delivered a few months later ("point release" known as "Version 1.1.").
- Agile, spiral, and iterative models are even more amorphous and phases can be freely interleaved and happen in parallel.

If this happens for the whole project, aspiring to a control of schedule and time for each activity and phase is utopic. This is an important limitation when, e.g., wanting to know the effects of specific techniques in the schedule or time devoted to an activity.

Capers Jones [11] also analyzes the measurement of time reaching these conclusions:

- Ambiguity in defining working periods (work days, work weeks, work months, and work years) which are usually applied to software projects. National and regional public holidays, vacation days, sick days, special days away due to weather, and non-work days for events such as travel, company meetings, etc. These exceptions are not always properly reflected and accounted in time sheets and whatever other method is used (if any systematic).
- Nobody really works every weekday but even a person may be physically at work for eight or nine hours a day and he/she would not be able or want to work solidly for eight hours every day. Coffee breaks, lunch, rest breaks, and social matters vary from one organization to another but you assume a good average up to two hours per day. And finally add days devoted to education, company

meetings, appraisals, interviewing new candidates, travel, and other activities that are usually not directly part of software projects. You can think in a total of 16 days per year.

- Finally ambiguity also comes from unpaid overtime applied to software projects.

Again, this makes very difficult to work in serious analysis of effects of decisions in schedule and time, which in the end means costs. Although many types of costs might be charged to a project, the main driver for expenses is the dedication of workers as software development is an activity very intensive in workforce. It is also clear that people involved in software development belong to very qualified categories and dedication is always expensive. This tight relationship between effort and cost leads to an assumed rule of substituting the typical units of cost (money) by units of effort: man-hours, man-month, etc.

When dealing with measuring costs, bad habits in software organizations do not help to our analysis [11]:

- Most corporate tracking systems for effort and costs (euros, work hours, person months, etc.) are incorrect and tend to omit from 30 percent to more than 70 percent of the real effort applied to software projects. Thus most companies cannot safely use their own historical data for predictive or analytic purposes.
- Productivity measurements based on human effort in terms of work hours or work months can be measured with acceptable precision if serious systems are implemented. But a problem with cost measures is that salaries and compensation vary widely from job to job, worker to worker, company to company, region to region, industry to industry, and country to country.
- Another impacting factor is the lack of generally accepted accounting practices for determining the overhead or indirect costs use for determining business topics and as input to contracts, outsource agreements, and return on investment (ROI) calculations. Even currency exchange in international projects and inflation in multiyear projects would distort the calculations.

Before proceeding with the definition of indicators and even with the determination of models for decision-making in the Iceberg project, we should perform a preliminary analysis on how targeted organizations manage the measurement of these factors.

5 EVALUATING QUALITY BY MEASURING ABSENCE OF IT: DEFECTS, INCIDENTS AND OTHER CONCEPTS

As stated in the introduction, there are many approaches to software quality but not all the organizations are in the level of maturity required to use all the models. The basic level and the most implemented one is the use of defects density measures, i.e. the rate between number of defects and the size of software. Defect-based metrics may represent a narrow view of quality, where quality is considered only as the lack of defects.

Defect density = number of known defects / product size

Defect density presents several problems related to the precision and real representation of the measured concept of software quality [1].

- No general consensus on what constitutes a defect. In fact, our work presented above in this section is oriented to standardize the definitions to be used in the project. If this agreement on how to measure is not clear, important problems appear. The main one is that different ways of measuring defects lead to not comparable results.
- No consensus about how to measure software size. Defect densities need to be calculated using consistent definitions of size to be comparable.
- Finding defects may tell more about the lack of quality and about the quality of defect finding and reporting processes, than it may tell about the quality of the product itself.
- Low defect rates are not a synonymous with quality in general. Some software fails do not necessary lead to failures perceived by users. And also programs or parts less used will be less prone to present defects.

Obviously many varieties as accounting indicators of software quality based on defects have been devised. The discipline of reliability has defined many indicators and measures. Many companies define their own varieties to reflect their view of what it is important to reach their quality goals. One is the Hitachi's System spoilage metrics defined as the ratio time to fix post-release defects/total system development time [12]. In fact, taken an illustrative list of some possible varieties from it, we can find the following ones with the acronyms (KNCLS: 1000 Non commentary source lines) and NCLOC (Non-commentary lines of code) [13]:

- Cumulative fault density
- Total serious faults found
- Mean time to close serious faults
- Total field fixes
- High-level design review errors per KNCSL
- Low-level design errors per KNCSL
- Code inspection error per inspected KNCSL

- Development test and integration error found per KNCSL
- System test problems found per developed KNCSL
- First application test site errors found per developed KNCSL
- Customer found problems per developed KNCSL

Defects have many aspects that should be considered relevant to be measured and analyzed with regard to the objectives of a strategy for quality assurance. The defects are inserted for some particular reason within the software artefacts; they have some impact and severity on the quality properties of the final product. They are detected at any specific time by noticing specific symptoms, using a detection technique which may or not include a support tool. Finally, the defects can be corrected or prevented by applying some kind of reasoning. Each one of these aspects may be relevant for the purpose of required analysis and also allow a categorization of defects.

For simplicity, Table 1 summarizes the key terms used within ICEBERG. The table provides a common vocabulary applicable to all projects' phases work (e.g., this glossary will be used to get the interview survey participants familiar with the overall goal of the project). It is intended to serve as a useful reference. To provide this list of terms, we have exploited the following standards and international glossaries:

a) the Standard IEEE 1044-2009 defined for software anomalies classification [14];
b) the glossary of the Information Technology Infrastructure Library [15];
c) the common vocabulary ISO/IEC/IEEE 24765:2010 applicable to all systems and software engineering work (prepared by ISO and liaison organizations IEEE Computer Society and Project Management Institute) [16];
d) the Standard Computer Dictionary ISO/IEC/IEEE 610:1990 [17].

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
MISTAKE	A human action that produces an incorrect result.
FAULT	A fault is a subtype of the super type defect. Every fault is a defect, but not every defect is a fault. A defect is a fault if it is encountered during software execution (thus causing a failure) but not if it is detected by inspection or static analysis and removed prior to executing the software.
ERROR	The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition
FAILURE	Termination of the ability of a product to perform a required function or its inability to perform within previously specified limits.
INCIDENT REFERENCE	Identification of the associated incident if the failure report was precipitated by a service desk or help desk call/contact.
PROBLEM	Difficulty or uncertainty experienced by one or more persons, resulting from an unsatisfactory encounter with a system in use. A problem may be caused by one or more failures, A failure may cause one or more problems.

RISK	The combination of the probability of an abnormal event or failure and the consequence(s) of that event or failure to a system's components, operators, users, or environment.
THREAT	Anything that might exploit a Vulnerability. Any potential cause of an Incident can be considered to be a threat.

Table 1. Key terms used within ICEBERG

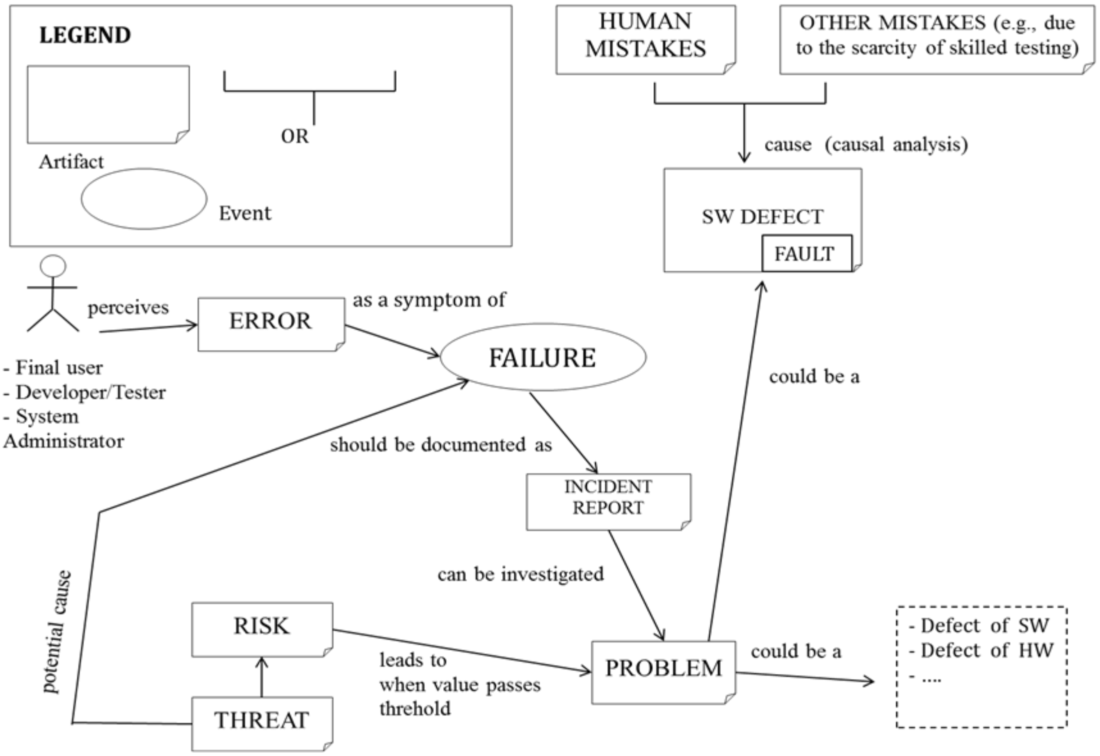

Figure 1: Dependencies among key terms of ICEBERG

Figure 1 describes the dependencies among the key terms listed in Table 1. The figure describes the event of a system failure, and how this event is typically managed. Shortly, a user (e.g., final user, a developer/tester or a system administrator) can perceive a system *failure*. Such no correct system behaviour should be documented as *incident report*. This latter could be further analyzed in order to understand the cause of the incident. Therefore, if a *problem* is raised, then further investigation will be performed. In particular, the nature of the failure is analyzed. In fact, a system failure could be, for example, raised either by hardware components or software components. If a software failure is recognized, then defect detection and removal techniques are adopted in order to analyze and fix the *defect* (i.e., the *fault* that has been the cause of the failure). Software defects could be introduced for different reasons, such as mistakes of the software developer/testers or the lack of skilled testing. Finally, it is also typically analyzed *the risk* that potential failure might occur in the future that will result in some negative consequences.

Defect classification schemes are designed to answer how, what and where to find software defects. Although there are several often cited classifications and even an IEEE standard [14], none of which have become a truly and broadly applied practice [18].

5.1 Defects Classifications Schemes

Defect classification schemes define a set of attributes and attribute values, where each attribute captures a specific aspect of the defect, e.g. symptom, type and injection mechanism.

The defect classification schemes more referenced in the literature can be classified into two categories according to their origins:

- Standard
- Industry
- Academy

5.1.1 Defects Classification Schemes Defined by Standards

There is one relevant standard that covers the classification of software defects: IEEE std. 1044. This standard was developed by IEEE in 1993. The latest release was launched in 2009. This work is based on the latest release. This standard provides a uniform approach to the classification of software anomalies, regardless of when they originate or where they are encountered within the project, product, or system life cycle. Classification data can be used for a variety of purposes, including defect causal analysis, project management, and software process improvement (e.g., to reduce the likelihood of defect insertion and/or to increase the likelihood of early defect detection) [14]. ***Table 2*** shows the attributes of the defects proposed by this classification scheme.

Attribute	Definition
Defect ID	Unique identifier for the defect.
Description	Description of what is missing, wrong, or unnecessary.
Status	Current state within defect report life cycle.
Asset	The software asset (product, component, module, etc.) containing the defect.
Artefact	The specific software work product containing the defect.
Version detected	Identification of the software version in which the defect was detected.
Version corrected	Identification of the software version in which the defect was corrected.
Priority	Ranking for processing assigned by the organization responsible for the evaluation, resolution, and closure of the defect relative to other reported defects.
Severity	The highest failure impact that the defect could (or did) cause, as determined by (from the perspective of) the organization responsible for software engineering.
Probability	Probability of recurring failure caused by this defect.

Effect	The class of requirement that is impacted by a failure caused by a defect.
Type	A categorization based on the class of code within which the defect is found or the work product within which the defect is found.
Mode	A categorization based on whether the defect is due to incorrect implementation or representation, the addition of something that is not needed, or an omission.
Insertion activity	The activity during which the defect was injected/inserted (i.e., during which the artefact containing the originated defect).
Detection activity	The activity during which the defect was detected (i.e., inspection or testing).
Failure reference(s)	Identifier of the failure(s) caused by the defect.
Change reference	Identifier of the corrective change request initiated to correct the defect.
Disposition	Final disposition of defect report upon closure.

Table 2: IEEE 1044 scheme attributes

This classification scheme has clear descriptions, exclusive mutually attributes and adaptable, covers a vast amount of defects information, it is designed at a more fine-grained that required and it has an orthogonal and hierarchical structure.

5.1.2 Defects Classification Schemes Defined in the Industry

There are some defects classification schemes used in the industry. However, they are for software defects and they do not consider the characteristics of conceptual schemas. The most relevant are the following

5.1.2.1 IBM Defects Classification Scheme

The IBM scheme is called Orthogonal Defect Classification (ODC) [20]. The first paper summarizing the full scheme was published in 1992.In this scheme a defect is classified across the dimensions shown in Table 3.

Attribute	Description
Defect type	Provides feedback on the development process (i.e. function, assignment, interface, checking, timing/serialization, build/package/merge, documentation, and algorithm).
Source	Type of code that is corrected (i.e. new, old, reused or fixed).
Impact	The resultant effect on the customer (e.g. capability, usability).
Trigger	Provides feedback on the verification process (e.g. testing, review, beta test).
Phase found	Defined on the development process activities (e.g. design, test).
Severity	IBM uses values between 1 and 4 where 1 is the highest signifying major outage, while 4 could be an annoyance.

Table 3: IBM scheme attributes

This classification scheme has almost the same properties as the previous scheme; the difference is that it only has an orthogonal structure and non-hierarchical [20]. It has been adopted by more and more organizations [21]. However, there are criticizes that the association between defect type and project phases is still an open question and that the distribution of defects types depends also on the processes and maturity of the company [18].

5.1.2.2 HP Defects Classification Scheme

This scheme was developed by HP's Software Metrics Council in 1986. The purpose of the scheme was to provide standard defect terminology that different HP projects and labs could use to report, analyse, and focus efforts to eliminate defects and their root causes [22]. In this scheme a defect is classified across the following three attributes (see Table 4).

Attribute	Description
Origin	The origin is the source of the defect (i.e. specifications/requirements, design, code, environmental support, documentation, other).
Type	A coarse-grained categorization of what is wrong. It is dependent on the value chosen for the Origin attribute.
Mode	It can be one of missing, unclear, wrong, changed and better way.

Table 4. HP scheme attributes

In this scheme the attribute *Type* is dependent on the value chosen for the attribute *Origin* (see Figure 2). This first requires analysis of when the defect was injected into the system before its type can be established. Therefore, its structure is semi-orthogonal. The HP scheme does not explicitly capture data about how a defect was detected. Additionally, there is no attribute available to identify which detecting mechanisms are effective in detecting particular defect types and investigate how severe defects can be identified [19].

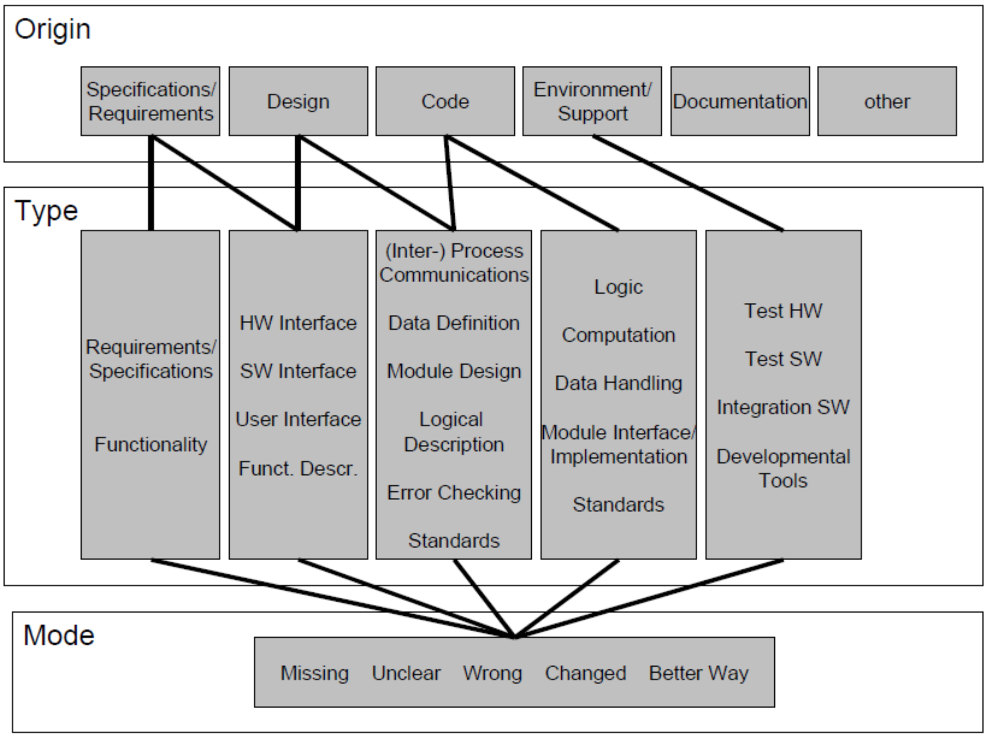

Figure 2: HP Defect Classification Scheme [22]

6 AN ADDITIONAL INDICATOR FOR EVALUATING QUALITY: SIZE

As commented in Section **Errore. L'origine riferimento non è stata trovata.**, evaluation and measurement of quality based on defect existence should be conceived in terms of defect density rather than as absolute numbers. This leads to the problem of having a trustable and consistent metric of size. Many different metrics of size have been devised during the existence of the software engineering discipline. We can mention that software size can be described with three attributes: i) length; ii) functionality; iii) complexity [24].

Below, a more detailed description of these attributes follows.

Length. The length is an attribute measured for the software specification, the design, and the code.

Depending on the used language, the size of the code is measured in several ways. The most used code measure is the number of lines of code (LOC). In order to count the lines, many schemas have been proposed. However, all existing approaches should basically provide guidelines to count the lines of a program by explaining how to handle typical code aspects (such as blank lines and comment lines), which must be considered during the code measurement. Big differences even from 1 to 5 could be found in the same piece of code depending on the criterion used to count LOC [24]. Other examples of atomic objects to count are executable statements, source instructions, characters or objects or methods. Even just the storage space in KB or MB could be used as indicator of size for executable code.

As far as the specification and the design is concerned, the adopted measures differ from the ones of code because of their different nature, which depends, for example, on the particular style, method or notation used. In fact, documents of specification and design usually combine different artefacts (like text, graph and mathematical symbol) that are incommensurate with respect to the length. Therefore, different measures must be used (e.g., the number of pages is used in industry to measure length for arbitrary types of documents).

Functionality. This attribute indicates the amount of function contained in delivered product or in a description of how the product is supposed to be.

Several methods have been introduced to measure the functionality of software products. We can mention, for example, three approaches: (i) Albrecht's function points; (ii) DeMarco's specification weight; and (iii) the COCOMO 2.0 approach to object points. These three approaches measure the functionality of specification documents, but they can be also applied to later life-cycle products in order to better refine size estimate, the cost or productivity estimate. Details on these approaches can be found in [24]. By far, all the measures in the style of Function Points with all the possible varieties from classical ones [25] to most recent ones like COSMIC[1] are the most used by organizations around the world. In Deliverable 2.1 additional approaches that deal with function points can be found (e.g., the work in [23]).

[1] http://www.cosmicon.com/

Complexity. This attribute is difficult to measure. We must distinguish in complexity of the problem (regarding the amount of resources required for an optimal solution) and complexity of the solution (regarding the amount of resources needed for the solution).

The complexity of the solution is related to the efficiency of the algorithm. The resource consumption (or computational cost) of the algorithm is measured. The complexity time of an algorithm is typically expressed by using the mathematical formalism, called big-O notation. This latter allows quantifying the amount of time needed for running the algorithm as a function of the size of the input. Normally this idea of complexity is not implemented in regular practice in industry.

Of course additional conceptions of complexity are available in literature as metrics for the different deliverables of projects but they are normally used for evaluating other attributes which might be related to the evaluation of quality rather than relating them to size. The work in [26], for example, investigates an approach for predicting the location of Aging-Related Bugs using software complexity metrics.

7 HOW TO DETERMINE MEASURES FOR ASSESSING THE THREE FACTORS OF THE IRON TRIANGLE

In order to determine which measures are normally used for assessing the three factors (i.e., cost, schedule and quality) in daily practice of organizations, we intend to conduct an interview-survey in which several customers and contacts of project's partners will be involved helping us to know how the basic factors are managed.

Although it could be assumed that regular practice of measuring these factors does not represent any difficulty to organizations, many problems may arise. Not all the organizations are measuring the basic factors in the same way, even they may name in different way the same concept or having different things with the same name. Some organizations are using ad-hoc tools, others commercial ones, others functionalities of general tools, etc. Some ones are collecting details and distinguishing among data from different tasks, others are collecting only data for the whole project and with few details, etc. In the end, the state of practice of the organizations will determine the feasibility of implementing the decision models existing in literature or adapting some of them to their situation, i.e. if they are not having the regular practice or maturity of collecting certain types of data with a specific level of detail, it would be impossible to propose them using certain specific frameworks or models for quality decisions as they won't have the necessary data to work with them or to make consistent decisions. In particular, as the Iceberg project wants to analyze how decisions on software quality during projects impact in the three factors of the Iron Triangle, we have to know what information would be available to a typical organization to implement a decision framework or model.

We also have to collect information on which types of decisions are normally made by managers or project leaders during the projects. As we want to propose a framework or a model for making decisions, it is necessary to know which SQA techniques and methods are normally adopted in projects to know which technical decisions related to them are logical to explore. It would be a non-sense to propose models where the impact of a specific method or technical strategy in a project (e.g. formal verification) if the organizations are not implementing it due to barriers like complexity, lacks of qualified staff, low maturity and absence of tradition or culture, etc. As a consequence, we have decided to include questions on SQA techniques and methods normally used in projects and on the types of decisions (i.e. the targeted factors in decisions) usually made. We are exploring the factors already analyzed in Deliverable 2.1 of project Iceberg to focus our effort of providing the most feasible and useful support for decisions.

Our study will be based on the method specified in [27] which is a formalized and repeatable process to document relevant knowledge on a specific subject area. We intend to conduct our survey in the main steps described in Table 5. We will refine these high level goals into more concrete sub-goals (i.e., short term objective) until it is possible to objectively measure their satisfaction. To this end, we will produce documents to be reviewed by university and industrial experts.

Step	Description
Step 1	Questionnaire Preparation
Step 2	Interaction with few industry representatives for questionnaire review

Step 3	Questionnaire Refinement
Step 4	Participants selection
Step 5	Interaction with participants in order to get them familiar with the overall goal of our study
Step 6	Questionnaire distribution
Step 7	Data collection and Analysis
Step 8	Addressing the Validity

Table 5: Interview-survey Steps

7.1 OBJECTIVE

We have identified these following research questions, which we intend to use (after the review of few industry representatives) in order to identify the measures of the iron triangle's factors.

RQ1: How cost and effort are measured? In which level of detail are data collected? Which mechanisms and tools are used for measuring, collecting and managing the data?

RQ2: How project duration is measured? In which level of detail are data collected? Which mechanisms and tools are used for measuring, collecting and managing the data?

RQ3: How quality is understood by people involved in projects? How are quality data collected? Which mechanisms and tools are used for measuring, collecting and managing the data?

RQ4: How projects or software size is measured? How are data collected? Which mechanisms and tools are used for measuring, collecting and managing the data?

RQ5: How the characteristics, culture, etc. of the organization are influencing the above measures?

RQ6: Which SQA techniques are normally implemented and used in the projects?

RQ7: Which factors are addressed in usual decisions during projects?

7.2 Questionnaire Design (Step 1)

By starting from our research questions, we have defined a questionnaire, which we have discussed with some industry representatives (see Step2 of Table 5). The questionnaire consists of two main parts. The first part is related to participants' personal data (e.g., company features, years of experience in the field), whereas the second part is related to all the aspects mentioned before, from the three factors of the iron triangle together with size to the SQA techniques and factors in decisions. We will use the data collected with the first part to summarize the distribution of the participants with respect to their working experiences and company features. In the additional document "Questionnaire-Survey", we have reported the complete questionnaire (http://bit.ly/IcebergBasicSurvey).

7.3 Validity

In this section we discuss the validity of our interview-survey. We have been inspired by validity threats proposed in [28] for an empirical study.

The following threats to validity and their solution have been identified.

Construct Validity. This threat is related to the high relation between the theory behind a study and its observation. Therefore, in order to assure high construct validity, we plan to adopt measures to conduct all steps of our work. The steps span from questionnaire preparation through respondents recruitment to data collection and analysis. Regarding the measures which will be used, we can remark the following points.

- We will assure a rigorous planning of the work with a solid protocol for questionnaire preparation (e.g., the questionnaire structure will facilitate the participants, and the data collection and analysis) and data collection and analysis.
- We will avoid mono-operation bias by planning several rounds of review and iteration with university and industry experts with different backgrounds and working experiences. We will discuss with them documents produced for the single steps of our work. We also intend to get these participants familiar with the overall goal of the work (and the single steps' results) in order to understand if there is a high relation between the theory behind our work and its achievements.
- We also intend to use measures for the participants' recruitment. We want, for example, to: (1) avoid mono-operation bias by selecting participants with different working experiences, (2) organize individual face-to-face meetings, based on a presentation illustrating the key concepts leading our study, to get the participant familiar, and (3) guarantee the anonymity and confidentiality in the results processing to avoid the evaluation apprehension.

Internal Validity. In order to assure the internal validity – referring to how well our work is done, we plan several rounds of review and iteration with university and industry experts. In order to avoid potential source of bias, we will, for example: (i) carefully balance our participations pool in according to their prevailing working experience and company features, and (ii) use a random-sample of the population in order to deal with the well know selection bias problem [29]. Furthermore, we also intend to address the ambiguously and poorly-worded questions issue [30], for example, by: (1) reviewing the questionnaire with industry experts, and (ii) performing individual face-to-face meetings with the participants during their work.

External Validity. This validity threat is related to the generalization of the results outside the scope of our work. We intend to take into account the project ICEBERG's sectors (e.g., banking, telecommunication and automotive), and provide results that are applicable in these domains. However, we also aim at generalize our conclusions by defining generic measures that could be adopted (without much effort) in different application domains.

Conclusion Validity. This last validity threat is related to the relation between the treatment and actual outcome we observe (i.e., why/how can we sure to draw correct conclusions?). In order to address this type of validity, we plan, for example, to: (1) adopt a solid protocol for data collection and analysis, (2) plan several rounds of review and iteration with university and industry experts, (3) assure a reliable treatment implementation by using the same treatment/procedure with all participants, and (4) assure the right heterogeneity of the respondents.

7.4 Results of the survey (Step 2)

We have discussed with few industry representatives our questionnaire. We have carried out the interview-survey among 9 experts. The participants were selected based on their affiliation and expertise.

Table 6 summarizes the results related to the level of details in which the respondents typically collect the data. The tables show the number of the answers (YES or NO). As shown in the tables, we have asked the respondents to outline (shortly discuss) the software phases in which the data are collected. By looking at the results in Table 6 and the respondents' comments, we have realized that they usually collect data for all the indicators (e.g., cost, time, quality, size) all along the software lifecycle. However, they did not specify particular phases, or better distinguish between development time and maintenance time.

Data Collection Level	Dimension			
	Cost	Schedule	Quality	Size
Project level (i.e. data referred only to the whole project)	3	0	6	7
Task/phase level (segmented data).	4	3	1	1
Software maintenance activities	2	6	2	1
TOTAL	9	9	9	9

Table 6: Collection Data Level

Table 7, Table 8, Table 9 & Table 10 summarize the results related to the metrics (or the data collection unit) collected for each factor respectively, i.e. Cost (Table 7), Schedule (Table 8), Quality (Table 9) and Size (Table 10). Table 11 summarizes how/if usually the respondents evaluate the monetary value. By looking at the results in Tables and the respondents' comments, we have realized that they usually deal with few software metrics, or defect (cost) data. Therefore, we have figured out that we need to work in this direction. We should better investigate, for example, (i) how code-level measurements and defect data are collected in the industry, and (ii) which is the effort required for collecting additional data (e.g., additional software metrics or particular cost factors, like the one for test cases generations or execution

Data Collection Unit	Schedule
week	1
days	5
hours	2
N/A	1
TOTAL	9

Table 7: Data Collection Unit for Schedule dimension

Data Collection Unit	Quality
incidents	1
defects	2
incidents AND defects	4
N/A	2
TOTAL	9

Table 8: Data Collection Unit for Quality dimension

Data Collection Unit	Size
multiple measures	2
Megabytes	1
FP	3
LOC and FP	2
N/A	1
TOTAL	9

Table 9: Data Collection Unit for Size dimension

Data Collection Unit	Cost

Person-hour	6
Person-days	3
TOTAL	9

Table 10: Effort Data Collection Unit for Cost dimension

Value Money Calculated	**Cost**
calculated costs for each employee	1
fixed rate per each effort unit	6
N/A	2
TOTAL	9

Table 11: Value Money Calculated per Cost dimension

Table 12 summarizes the results related to the systems used to collect data for each factor (i.e., cost, schedule, quality, size). Table 13 lists tools that are usually used by the respondents. By looking at the results in Tables and the respondents' comments, we have realized that commercial tools (typically open-source) are often used for data collections. In particular, we have realized that they use tools for source code metrics evaluation (e.g., Sonar), and bug tracking (e.g., JIRA). We should better investigate, for example, (i) which is the effort required for using in industry a tool for software metrics evaluation, and (ii) which additional data could be collected with the used tools.

System used to collect data	**Dimension**			
	Cost	**Schedule**	**Quality**	**Size**
Specific solution from general commercial tool (project management tools, etc.)	5	4	3	1
Specific solution from general tools (database, programmed solution, etc.)	2	3	2	3
Ad-hoc solutions (Excel, timesheet, etc.)	1	0	2	4
Specific commercial solution	1	1	2	1

Table 12: System used to collect data

	Dimension			
	Cost	**Schedule**	**Quality**	**Size**

Tools	MS Project Server, JIRA, SAP PPM, OnePoint (planning, allocation) custom ERP (corporate DB) MS Excel (Analysis), SAP	MS Project Server, JIRA SAP PPM, OnePoint (planning, allocation) Cusom ERP (corporate DB) MS Excel (Analysis), Microsoft Project Manager	JIRA, iTestman (custom/adhoc tool) Trac (open source) Redmine (open source) Customer tool, IBM Rational	Excel, JIRA, SonarQube, Word documents

Table 13: Tools used to collect data

Table 14 summarizes QA techniques/approaches, which the respondents typically consider in their projects. Table 15 lists factors, which affect the QA decisions. Tables just show the number of answers (YES or NO) per QA technique/approach (or factor). By looking at the results in Tables and the respondents' comments, we have realized that the testing is a typical activity in the industry. Moreover, tools and techniques for automating testing activities are also usually used. For example, Selenium test suite[2] for testing automation is adopted, or the TestLink[3] web tool is used. Therefore, we have realized that we should better investigate, for example, (i) which are the main features of these tools adopted for the testing, (ii) how these tools could be integrated with other tools (e.g., the ones for source code metrics evaluations), and (iii) which is the effort required for adopting a new testing tools in the industry.

QA activity/techniques	Total (N=9)
Testing	9
Testing automation	6
Static code analysis	9
Software metrics tools	7
Software inspections	8
Software configuration management	7

Table 14: QA activities/techniques typically considered by the industrial experts

Factors	Total (N=9)
Human factors	5

[2] http://docs.seleniumhq.org/docs/01_introducing_selenium.jsp

[3] http://sourceforge.net/projects/testlink/

Organizational Factors	5
QA processes and techniques	8
Development processes and techniques	7
Technology selection	7
Environment and support	4

Table 15: Factors involved in QA decisions

8 EXISTING CATEGORIES OF APPLICABLE DECISION MODELS

Once analyzed the result of the pilot survey on basic indicators for decisions in software projects, we intend to review the state of the art in order to analyze which existing approaches for quality decision-making are adequate to the state of practice in organizations and then which indicators and metrics are required or can be used, analyzing at the same time the availability of collection of such a list of data within the regular practice of the organizations As a consequence, in this section, our goal is twofold: (i) presenting a holistic overview of feasible quality decision-making approaches that have been reported in literature, and (ii) categorization of indicators/metrics related to such quality decision-making models.

8.1 QUALITY DECISION-MAKING APPROACHES

Several research efforts have been devoted to the definition of quality decision-making approaches in each phase of the software lifecycle. Different techniques have been introduced in order to, for example: (1) analyze the impact of architectural decisions on system quality (e.g., the Architecture Tradeoff Analysis Method [31]), (2) estimates costs, (short-term and long-term) benefits and uncertainty of architectural design decision (e.g., the CBAM method [32]), (3) derive test plans from requirements [33], or use architectural artefacts (like software architecture specification models, architectural design decisions, architectural documentation) in testing the implementation of a system (e.g., test cases, test plans, coverage measures) and executing code-level test cases to check the implementation [34], and (4) select testing techniques according to the features of the software to test [35].

Optimization techniques have largely been used to automate, for example: (i) the testing process (e.g., the one of the mutation testing [36]); (ii) the search for an optimal architecture design with respect to a (set of) quality attribute(s) (see, survey [37]); or (iii) the adaptation of a software architecture (both its structure and behaviour) with non-functional attributes tradeoff (e.g., [38]). Existing approaches are basically based on simple optimization models (e.g., in [38] the adaptation cost is minimized) or multi-objective optimization models (for example, for test case generations [39] or cross-project defect prediction [40]) maximizing a set of objectives (e.g., maximizing data flow coverage and minimizing the size of the test set [41], or minimizing adaptation costs and system probability of failure [42]).

However, decisions are not only made at the application level, but also at the project management level (i.e., schedule/time-related decisions are made). Research efforts have been spent for software development estimation, by using, for example a statistical model for managing selection bias effects [43], or the soft system methodology to establish a benchmark for managing cost overruns in software projects [44]. Other papers have focused on project staffing and scheduling using different approaches such a Mixed-Integer Linear Program (MILP) [45], a hybrid MILP/constraint programming benders decomposition algorithm [46], and a MIP-based approach [47].

Emerging computing application paradigms require systems that are not only reliable, compact and fast, but which also optimize many different competing and conflicting objectives, like response time, throughput and consumption of resources [48]. Any combination of quality decisions may have a considerable impact on cost, time/schedule decisions. Therefore, a major issue in this direction is that decisions at a single system level (i.e., the application or the project management level) cannot be analyzed separately, because they (sometimes adversely) affect each other. These evaluations can suffer of large elapsed time when the search space size increases. In such cases, the complete enumeration of possible alternatives results inefficient. The adoption of these SBSE search methodologies (e.g., genetic algorithms, evolutionary algorithms and other metaheuristics) has already been proposed as a viable solution both for the application level and for the project management level.

For example, for quality decisions, they have been used in order to support: (i) the generation of software test cases [49], [50], and develop testing tools such as AUSTIN for unit testing C programs [51], (ii) the large-scale QoS-aware service compositions [52], [53], and automate the search for an optimal architecture design based on functional and non-functional requirements tradeoffs [42], and (iii) the distributed system's allocation of software components to hardware nodes (i.e., deployment architecture) while guaranteeing a specific level of QoS properties [54].

8.2 Schedule/Time decision-making approaches

SBSE techniques have also been largely applied in problems in software project management (see, e.g., [55]). A quite extensive list of these approaches can be found in [56].

As mentioned in [56], research efforts have been devoted for project scheduling and resource allocation. However, all these approaches basically provide guidelines to plan projects. Their primary input is represented by information about (i) work packages (e.g., cost, duration, dependencies), and (ii) staff skills. Shortly, as described in [56], they process these input information and produce the results, which consist of an optimal work package ordering and staff allocation. They are guided by a single or multi-objectives fitness function which it is typically minimized, for example: the completion time of the project, or the risks to associate to the development process (e.g., delays in the project completion time, or reduced budgets available).

As outlined in [56], SBSE methodologies have been also applied to build effort estimation models or to enhance the use of other estimation techniques (e.g., genetic programming has been used in [57] to validate the component-based method for software sizing, and a tabu search approach has been adopted in [58] to estimate software development effort). An overview of existing approaches is provided in [56], and their advantages/limitations and open challenges are also outlined. These approaches could be exploited, for example, (i) to support the choice of a reliable measure to compare different estimation models; or (ii) to investigate prediction uncertainty and risk of inaccurate prediction by means of using sensitivity analysis or multi-objective optimization (they only have been used to obtain exact prediction, i.e., one point estimate for a project).

Some search-based SE papers for project managements are focused on the problem of process risk (e.g., [59]) and the product risks (e.g., [60]). "Risks to the product concern the possibility that there may be flaws in the product that make it less attractive to customers, while process risks concern the problems that may cause delays in the project completion time, or reduced budgets available forcing compromise."[56].Finally, the overtime planning is also considered in [61]. Specially, this work introduces a multi-objective decision support approach to help balance project risks and duration against overtime.

Even though interest in the exploration of the SBSEs potential as a means for quality decision making and software project management has also grown rapidly, there are still big research challenges to be addressed.

8.3 SCHEDULE/TIME AND QUALITY DECISION-MAKING APPROACHES

Coordination aspects between the application level and the project management level have already been exploited. We can remark the following points.

(i) *"Build-or-buy" decisions in a software architecture, system delivery time constraints, and testing have been considered together*. In [62], a framework for supporting "build-or-buy" decisions in a software architecture has been introduced. Specifically, this work presents a non-linear cost/quality optimization model based on decision variables indicating the set of architectural components to buy and to build in order to minimize the software cost under reliability and delivery time constraints. The model can be ideally embedded into a Cost Benefit Analysis Method to provide decision support to software architects. Such formulation involves further variables representing the amount of unit testing to be performed on each in-house developed component.

(ii) *Reliability and costs together have been considered in different contexts*, for example to provide guidelines in (1) evaluating the effort spent to test the software, deal with the resource allocation during the test process or quantify the costs of service failure repair/mitigation actions (see, e.g., [63], [64], [65], [66], [67], [68], [69], [70], [71], [72], [73], [74], [75] some of which are detailed below), or (2) comparing the costs of defect-detection techniques [76].

In [67] it is formulated a reliability constrained cost minimization problem, where the decision variables represent the component failure intensities. Specifically, in order to represent the dependency of the component cost on the component failure intensity (i.e., the cost to reach a specific failure intensity) three different types of cost functions (i.e. linear, logarithmic exponential, inverse power) are exploited. This model works after the components have been chosen; as its solution provides insights about the failure intensities that the (selected) components have to attain to minimize the system cost.

Resource allocation during the test process in modular software systems is dealt, for example, in [74]. Specifically, this work presents a framework for performing resource allocation (budget and time) during the test process of a software system. The framework exploits a model developed with the goal of finding the maximum reliability of the software system while satisfying a budget limit on the total test cost and minimum reliability of components. The paper assumes that a software system has already been specified, designed and coded.

The work in [75] presents an approach for service selection taking into account costs and reliability requirements. In particular, it defines a set of optimization models that allow quantifying the costs of service failure repair/mitigation actions aimed at keeping the whole system reliability over a given threshold.

(iii) *Models for achieving product and process improvement have been introduced*. The goal of these models is to ensure a capable process, i.e., a process that produces a significantly reduced number of exploitable defects (see, the work in [77] that provides advice for those making a business case for building software assurance into software products during software development). Examples of process improvement models include the Software Engineering Institute's Capability Maturity Model Integration (CMMI) framework, along with the retired Capability Maturity Model for Software (Software CMM). These models address process capability by assessing the presence, or absence, of proxies (i.e., essential practices that are generally considered to ensure against defects).

Research efforts have also been spent in order to deal with the automated selection and configuration of methods and tools. The work in [78] analyzes challenges of managing engineering tool variability in context of engineering project environment configurations. For example, best-practice method support (e.g., the QATAM technique for the evaluation of QA strategies and their tradeoffs in the context of a software process) is discussed. The work also presents a conceptual approach using semantic modeling of project requirements and tool capabilities.

(iv) *How human and organizational factors influence software quality practices and productivity has been investigated*. The work in [79] conducts a survey to understand which is the situation of real testing practice and which factors mainly related to professionals (attitude, training or similar items) are having a real influence in software quality in terms of the perception of participants.

8.4 Software Quality Evaluation

As remarked in Deliverable 2.1, several important product/process quality standards (such as ISO/IEC 15504, ISO/IEC 29119, and ISO 9126) have been introduced. By looking at the results of our questionnaire review, we have realized that these quality methodologies would require much effort and skill to be applied in the industry. In the contrast, we have figured out that practitioners usually deal with software metrics and defect data. Attributes of software quality, such as defect density, should be easy collected and evaluated. Therefore, we have realized that we should focus on utilizing software metrics, such as code-level measurements and defect data. We will investigate how defect prediction methods are able to predict the poor-quality of industrial projects, and how their adoption can be facilitated and automated.

8.4.1 Metrics as Quality Indicators

Several research efforts have been devoted to the definition of defect prediction methods able to predict the poor-quality of program modules (e.g., [80]). These approaches utilize software metrics and defect data collected during the software development process. Their efficacy is, therefore, influenced by the relevance between software metrics and fault data. The modules predicted to be fault-prone will receive more inspection and testing, thereby improving their quality.

The *accuracy* and the *granularity* are two important qualities of software fault prediction algorithms [81]. The accuracy represents the degree to which the algorithm correctly identifies future faults. On the contrary, the granularity specifies the locality of the prediction. As remarked in [81], typical fault prediction granularities are: (i) the executable binary [82]; (ii) a module (often a directory of source code) [83]; (iii) or a source code file [84]. A directory level of granularity, for example, means that predictions indicate a fault will occur somewhere within a directory of source code. As stated in [81], the most difficult granularity for prediction is the entity level (or below), where an "entity" is a function or method. In [81], for example, Kim at al. developed an algorithm that, in their experimental assessment on seven open source projects, is 73%-95% accurate at predicting future faults at the file level and 46%-72% accurate at the entity level with optimal options. In Section 8.4.2, we better discuss the Kim at *al*. approach.

The literature contains a wealth of software metrics proposed for software fault prediction. In fact, software metrics may be used in prediction models to improve software quality by predicting fault location [85]. The work in [85] presents the results of a systematic literature review in software fault prediction. Specifically, it gives an overview of the current state-of-the-art software metrics in software fault prediction. They categorized existing studies according to the metrics used in the following manner:

- *Traditional:* size (e.g. LOC) and complexity metrics (e.g.McCabe [86]).
- *Object-oriented*: coupling, cohesion and inheritance source code metrics used at a class-level (e.g. Chidamber and Kemerer [87]).
- *Process:* process, code delta, code churn, history and developer metrics. These metrics are usually extracted from the combination of source code and repository, and they require more than one version of a software item.

The paper [85] provides a deep analysis of which metrics are, and which are not, significant fault predictors. Moreover, the authors assessed the data sets used in the studies, the software development life cycle phases in which the data sets are gathered, and the context in which the metrics were evaluated. However, the results of this systematic review can be summarized using the following authors' statements.

- Object-oriented metrics (49%) were used nearly twice as often compared to traditional source code metrics (27%) or process metrics (24%).

- Chidamber and Kemerers (CK) object-oriented metrics were most frequently used. According to the selected studies, there are significant differences between the metrics used in fault prediction performance.
- Object-oriented and process metrics have been reported to be more successful in finding faults compared to traditional size and complexity metrics. Process metrics seem to be better at predicting post-release faults compared to any static code metrics.

Different prediction approaches have been introduced by relying on diverse information (e.g., on source code metrics, process metrics or previous defects). The efficacy of defect prediction models is influenced by relevance between software metrics and fault data [88]. A typical problem often encountered by software practitioners is the presence of excessive metrics in a training data set. Research effort has been devoted in this direction, namely approaches for supporting the choice of the most important metrics (features) prior to the model training process have been introduced (see, for example, [88], [89], or the *goal-question-metric method* [90][91] described here below).

The Goal-Question-Metric Method. The Goal-Question-Metric (GQM) method [90] [91] proposes a measurement method for assessing or improving the quality of entities like products, processes or people (see Figure 3 for an example). It starts with a set of business goals and the goals are progressively refined through questions until we obtain some metrics for measurement. The measured values are then interpreted in order to answer the goals. Existing approaches choose a quality model from those that exist so as to generate the business (or the primary) goals of the GQM formulation for any individual product or process.

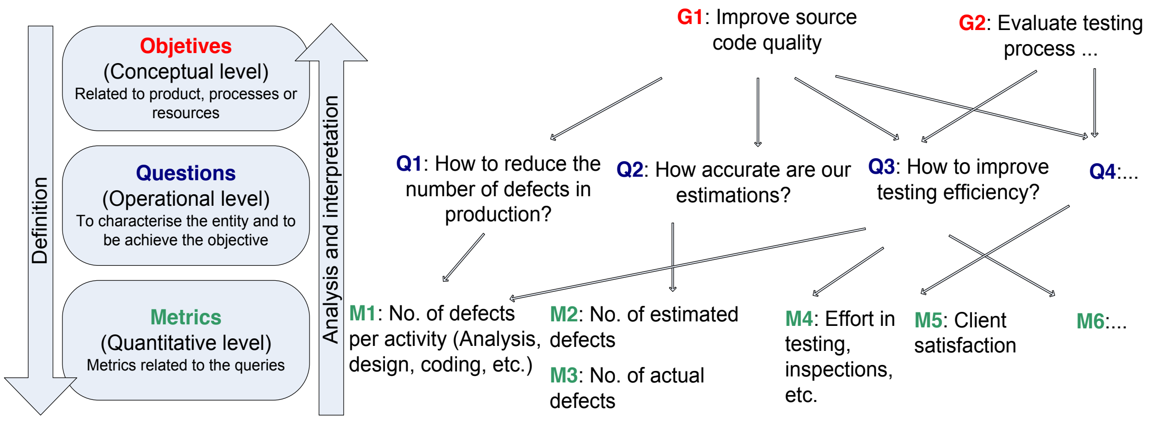
Definition
Objetives
(Conceptual level)
Related to product, processes or resources
Questions
(Operational level)
To characterise the entity and to be achieve the objective
Metrics
(Quantitative level)
Metrics related to the queries
Analysis and interpretation
G1: Improve source code quality
G2: Evaluate testing process ...
Q1: How to reduce the number of defects in production?
Q2: How accurate are our estimations?
Q3: How to improve testing efficiency?
Q4:...
M1: No. of defects per activity (Analysis, design, coding, etc.)
M2: No. of estimated defects
M3: No. of actual defects
M4: Effort in testing, inspections, etc.
M5: Client satisfaction
M6:...

Figure 3. Levels of the Goal-Question-Metric Method and an example

A quite extensive list of defect prediction approaches can be found in [92] [93]. However, all these approaches basically provide guidelines to predict defects in source code by exploiting the usefulness of elementary metrics or previous defects. They have the following common steps that can be iterative and overlapping.

- ***Step 1.*** The metrics evaluation is accomplished. Depending on the adopted type of metrics (e.g., object-oriented metrics or "traditional" product metrics, like number of lines of code, McCabe complexity), different computing approaches are used.

- ***Step 2.*** The relationships between the values of the metrics and the numbers of bugs found in the system (e.g., in the classes) are discovered. Well-known statistical methods (e.g., logistic and linear regression) have been largely adopted to validate the usefulness of the metrics to identify defective classes. Basili et *al*. in [94], for example, validate object-oriented design metrics as quality indicators by using logistic regression technique [95]. In the contrast, Gyimóthy at *al*. in [96], besides using regression methods (logistic and linear regression), also employed machine learning techniques to validate the usefulness of object-oriented metrics for fault-proneness prediction on open source software.

In order to validate the metrics' usefulness for fault-proneness, the output of the previous step is analyzed. Specifically, the values obtained are checked against the number of bugs found in the system (e.g., in [96] the values of the object-oriented metrics of the open source Web and e-mail suite called Mozilla are checked against the number of bugs found in its bug database called Bugzilla[4]).

8.4.2 Metrics Evaluation

Several research effort has been devoted to the definition of methods and tools able to evaluate software metrics. In the following we discuss: (i) the main kinds of software metrics, and (ii) significant defect prediction approaches. We also discuss existing tools both for the acquisition and presentation of the values of metrics. These tools bring important advantages [98], such as the reduction of metric calculation errors, thus achieving greater accuracy in their values.

A metrics should clarify what attributes of the software that are going to be measured and how we go about measuring those attributes [99][100][101]. So, they should be meaningful and related to the product. Metrics can be evaluated theoretically or empirically. On the one hand, [100] describe a list theoretical features that metrics (direct and indirect) must hold to be valid.

For *direct metrics*, which are the ones that involves no other attribute or entity (length, duration of testing process, number of defects…), those properties are:

1. For an attribute to be measurable, it must allow different entities to be distinguished from one another.
2. A valid metric must obey the representation condition.
3. Each unit contributing to a valid metric is equivalent.
4. Different entities can have the same attribute value.

For *indirect metrics*, when direct metrics are combined (ex., programmer productivity = LOC/persons month of effort, etc):

1. The metric should be based on an explicitly defined model of the relationship

[4] http://www.bugzilla.org/

between certain attributes.

2. The model must be dimensionally consistent.
3. The metric must not exhibit any unexpected discontinuities.
4. The metric must use units and scale types correctly.

The *representation condition*, as described by [102], asserts that a measurement mapping *M* must map entities into numbers and empirical relations into numerical relations in such a way that the empirical relations preserve and are preserved by the numerical relations.

On the other hand, empirical methods provide corroborating evidence of their validity. Using statistical and experimental techniques assess the usefulness and relevance of the metrics [99] [103].

Source code metrics. Many approaches in the literature use this kind of metrics (see, for example, [94] and [96]). As remarked in [85], most of the works exploited the suite for object oriented design (also named CK metrics) introduced in [87]. As shown in ***Table 16***, the CK metrics suite involves 6 metrics calculated for each class. These metrics have been calculated and validated in several different ways, some of which are detailed below.

Metric	
WMC	Weighted Method Count
DIT	Depth of Inheritance Tree
RFC	Response For Class
NOC	Number Of Children
CBO	Coupling Between Objects
LCOM	Lack of Cohesion in Methods

Table 16: CK metric suite [87]

In [94], Basili et *al*. have used eight projects developed by using a sequential life cycle model, a well-known OO analysis/design method. The projects were written by students in C/C++. Basili et *al*. have slightly adjusted some of CK metrics in order to reflect the specificities of C++. Based on empirical and quantitative analysis, they have argued that several of CK metrics appear to be useful to predict class fault-proneness during the early phases of the life-cycle. Moreover, they have also figured out that, on their data set, CK metrics are better predictors than “traditional” code metrics, which can only be collected at a later phase of the software development processes. *GEN++* [104] was used to extract CK metrics directly from the source code of the projects delivered at the end of the implementation phase.

These CK metrics, slightly modified to reflect the specificities of C++, have also been used by the work in [96]. In addition, this work considered the LCOMN metric (i.e., the Lack of Cohesion on Methods allowing Negative value), and used the well-known lines of code metric (LOC). The goal of this paper was to calculate and validate these metrics for fault-proneness detection of the source code of Mozilla. In particular, the source code of Mozilla has been analyzed by using the *Columbus* framework [105].

Shortly, Columbus is a reverse engineering framework developed in cooperation between the University of Szeged, the Nokia Research Center and FrontEndART [106]. Columbus has been developed to define several fundamental building blocks for the use in reverse engineering processes, and as such it can be an important player in the studies conducted at the workshop for Empirical Studies in Reverse Engineering. Columbus provides support for the extraction in general, and a common interface for other reverse engineering tasks as well. The framework makes available all the necessary components (i) to perform the analysis of arbitrary C/C++ source code and, (ii) to present the extracted information in any desired form.

The authors in [107] also compared the CK metrics with additional object-oriented metrics, and LOC metric. Five open source systems were used to validate the findings (Eclipse JDT Core, Eclipse PDE UI, Equinox framework, Mylyn, and Apache Lucene). ***Table 17*** lists all source code metrics used.

Type	Metric Name	Definition
CK	WMC	Weighted Method Count
CK	DIT	Depth of Inheritance Tree
CK	RFC	Response For Class
CK	NOC	Number Of Children
CK	CBO	Coupling Between Objects
CK	LCOM	Lack of Cohesion in Methods
OO	FanIn	Number of other classes that reference the class
OO	FanOut	Number of other classes referenced by the class
OO	NOA	Number of attributes
OO	NOPA	Number of public attributes
OO	NOPRA	Number of private attributes
OO	NOAI	Number of attributes inherited
OO	LOC	Number of lines of code
OO	NOM	Number of methods
OO	NOPM	Number of public methods
OO	NOPRM	Number of private methods
OO	NOMI	Number of methods inherited

Table 17: Class level source code metrics used in [107]

To evaluate the metrics, the authors used the *Moose* suite.[5] Shortly, Moose provides a platform for software and data analysis. In particular, it offers multiple services ranging from importing and parsing data, to modelling, to measuring, querying, mining, and to building interactive and visual analysis tools. The authors in [107] used the *Moose* tools to read FAMIX models and to compute a number of source code metrics. In fact, they

[5] Moose is available at http://www.moosetechnology.org

derived an object-oriented model of the system source code according to FAMIX, a language independent meta-model of object oriented code [109].

The work in [108] defined what is called the MOOD suite of Object Oriented metrics.

Type	Metric Name	Definition
OO	MHF	Method Hiding Factor
OO	AHF	Attribute Hiding Factor
OO	MIF	Method Inheritance Factor
OO	AIF	Attribute Inheritance Factor
OO	POF	Polymorphism Factor
OO	COF	Coupling Factor

Table 18: MOOD suite of Object Oriented metrics defined [108]

The ***Specialisation Index per Class (SIX)*** metric [97] measures the extent to which subclasses override (replace) behaviour of their superclasses.

$$\mathit{Specialization\ index} = \frac{\text{Number of overriden methods} \cdot \text{Class hierarchy nesting level}}{\text{Total number of methods}}$$

This formulation weights more heavily overrides that occur farther down the inheritance tree since these classes should be more specialised and less likely to replace base behaviour.

Other open source tools for measuring the Internal Quality of Java software products
The work in [98] presents the results of a study of the state-of-the-art open source software tools that automate the collection of metrics, particularly for developments in Java. Specifically, the study is focused on Internal Quality metrics of a software product and software tools of static code analysis that automate measuring these metrics. The static analysis of the code is defined as a set of analysis techniques where the software studied is not executed (in contrast to the dynamic analysis), but analyzed. Therefore, this type of analysis will allow obtaining Internal Quality metrics, as it does not require a software in use to be measured. To perform this comparative analysis of tools, the authors in [98] conducted a systematic literature review.

Table 19 shows the information extraction template for cataloging and comparing tools, which the authors used.

Attributes	Dominion
Internal Quality models supported	{ISO 9126, ISO 25010, SQUALE, SIG}
Metrics implemented	{Complexity, CK, code size, comment size, coding convention violations, code smells, duplicated code, dependencies}
Functional features covered	{Data acquisition, analysis of measures, data presentation}
Year of first version	Year
Year of last version	Year

Table 19: Characterization scheme for the description of tools used in [98]

Table 20 summarizes the metrics implemented by tools. As shown in table, code smells metrics are the most covered by the tools (7 tools), closely followed by complexity and code size metrics (6 tools). Moreover, the authors figured out that most tools only implement a small set of metrics (since they are highly specialized), except for Sonar and Squale tools that cover all categories and become the most complete tools in relation to this feature. The authors pointed out that most of tools automate the calculation of Internal Quality metrics (data acquisition), being code smells, complexity and code size the most common ones. They asserted that the Sonar and Squale tools are capable of gathering data for all categories of metrics, while the other tools are more specialized in a limited set of metrics.

As shown in table, the authors mainly analyzed: (i) which are the Internal Quality models implemented by the tools, i.e. the possible relationship between metrics and quality models is investigated, (ii) the metrics implemented, and (iii) the functional features covered. This latter attribute involves the three main tasks that metric tools must perform (i.e. *Data acquisition, Analysis of the measures, Data presentation*).

- The *Data acquisition* is related to the set of methods and techniques for obtaining necessary data for measurement.

- The *Analysis of the measures* is related to the ability to store, retrieve, manipulate and perform data analysis.

- The *Data presentation* is aimed to provide formats to generate the obtained documentation. Examples of possible representation are tables and graphs or exporting files to other applications.

Table 20 summarizes the metrics implemented by tools. As shown in table, code smells metrics are the most covered by the tools (7 tools), closely followed by complexity and code size metrics (6 tools). Moreover, the authors figured out that most tools only implement a small set of metrics (since they are highly specialized), except for *Sonar* and *Squale* tools that cover all categories and become the most complete tools in relation to this feature. The authors pointed out that most of tools automate the

calculation of Internal Quality metrics (data acquisition), being code smells, complexity and code size the most common ones. They asserted that the Sonar and Squale tools are capable of gathering data for all categories of metrics, while the other tools are more specialized in a limited set of metrics.

	Complexity	**CK**	**Code size**	**Comment size**	**Coding convention violations**	**Code smells**	**Duplicated code**	**Dependencies**	Total
Jdepend								x	1
JCSC	x		x		x	x			4
QALab									0
CKJM		x							1
Panopticode	x		x	x				x	4
Same							x		1
FindBugs						x			1
JavaNCSS	x		x	x					3
PMD/CPD						x	x		2
Xradar	x	x	x	x					4
Checkstyle					x	x			2
Sonar	x	x	x	x	x	x	x	x	8
Classycle								x	1
Jlint						x			1
Sonar Plugins									0
Squale	x	x	x	x	x	x	x	x	8
TOTAL	6	4	6	5	4	7	4	5	

Table 20: Metrics implemented by tools [98]

Shortly, *Sonar*[6] is an open platform to manage code quality. Sonar is mainly composed by a maven plugin that performs static analysis and a web application that stores metrics in a database and presents them. In particular, Sonar is able to gather metrics in all categories: (i) code size (e.g., Lines of Code, Classes); (ii) comment size (e.g., Density of comment lines); (iii) duplicated code (Density of duplicated lines and some others related), (iv) complexity (e.g., Average complexity by method, Average complexity by class, Average complexity by file), (v) coding convention violations and code smells (e.g., Violations), dependencies (e.g., Package tangle index, Package cycles, Package dependencies to cut, File dependencies to cut); and (vi) CK metrics (LCOM and RFC). Note that Sonar is one of the tools used by our questionnaire respondents (see Section 6.4).

[6] http://www.sonarqube.org/

Squale (Software QUALity Enhancement)[7] is a qualimetry platform that allows to analyze multi-language software applications. Squale is mainly composed by a web application that presents metrics (SqualeWeb), and a batch process (developed in Java) that performs the analysis of source code. Squale is able to gather metrics in all categories: (i) complexity (CCN and summation of CCN per class); (ii) CK metrics (DIT, LCOM, Ca, RFC, Ce); (iii) code size (NCSS, NOM and number of classes); (iii) comment size (number of comment lines); (iv) coding convention violations and code smells; (v) duplicated code (number of duplicated lines); (vi) and dependencies (number of dependency cycles and Distance from the main sequence (D)).

Table 21 lists the functional features of the tools. In particular, the authors have figured out that there are three complete tools (i.e., Xradar, Sonar and Squale), which perform the data acquisition, analysis and presentation. They also realized that these three tools are relatively new and are based on more mature tools for metric acquisition.

	Data acquisition	**Analysis of measures**	**Data presentation**	Total
Jdepend	x			1
JCSC	x			1
QALab			X	1
CKJM	x			1
Panopticode	x		x	2
Same	x			1
FindBugs	x			1
JavaNCSS	x			1
PMD/CPD	x			1
Xradar	x	x	x	3
Checkstyle	x			1
Sonar	x	x	x	3
Classycle	x			1
Jlint	x			1
Sonar Plugins		x	x	2
Squale	x	x	x	3
TOTAL	14	4	6	

Table 21: Functional features covered by tools [66]

Shortly, Xradar[8] is an open extensible code report tool currently supporting all Java based systems. Xradar is capable of measuring complexity metrics (CCN), CK (WMC, DIT, CBO, RFC, LCOM, Ce, Ca), code size (NCSS, NOM, number of classes), comment size (number of javadocs, number of single-line comments, number of block

[7] http://www.squale.org/

[8] http://xradar.sourceforge.net/

comments), coding convention violations, code smells, duplicated code (duplicated lines, duplicated blocks, duplicated tokens) and dependencies (number of cycles, I, D).

More details on Sonar, Square, and Xradar tools (and other tools) can be found in [98].

Change Metrics Research effort has been devoted for analyzing the predictive power of process related software metrics. One of the most known work that presents a comparative analysis of the efficiency of change metrics and static code metrics for defect prediction is the approach of Moser et *al*. [110]

In particular, Moser et *al*. used a public data set created by Zimmermann et *al*. [111]. The data set includes a large number of static code metrics (198 attributes) and pre- and post-release defects for the Eclipse releases 2.0, 2.1, and 3.0. It is available for download in the PROMISE repository.[9] In fact, Zimmermann et *al*. mapped defects from the bug database of Eclipse (one of the largest open-source projects) to source code locations (files) by using some heuristics based on pattern matching. Moser et *al*. also extracted 18 change metrics from the Eclipse CVS repository[10] and annotated the original data set with them. They analyzed the relationship of change and code metrics with post-release defects only at a file level. In particular, they investigated whether or not a file is defect-free.

Table 22 summaries the augmented data set and the class distribution, i.e., the number of defective and defect free files.

Release	#Files	Metrics	Defect Free	Defective
2.0	3851 (57%)	31code metrics and 18 change metrics	2665	1186
2.1	5341 (68%)		4087	1254
3	5347 (81%)		3622	1725

Table 22. Summary of the Eclipse data used in the study [110].

As shown in ***Table 23***, they considered only a subset of 31 metrics which were used by Zimmermann et *al*. for defect prediction at a file level. Zimmermann et *al*. obtained promising results for predicting the presence of defects in packages, but only fair results for classifying single files as defect free respective defective [110]. Moser et al compared the model used by Zimmermann et *al*. with the one based on change data and a combination of the two. ***Table 23*** shows the change metrics used in the Moser et *al*. approach. They derived this set of metrics by exploiting related works (e.g., [112], [82] and [84]).

Metric Name	Definition

[9] http://promisedata.org/repository

[10] A versioning system (CVS) enables the handling of different versions of files in cooperating teams.

REVISIONS	Number of revisions of a file
REFACTORINGS	Number of times a file has been refactored
BUGFIXES	Number of times a file was involved in bug-fixing
AUTHORS	Number of distinct authors that checked a file into the repository
LOC_ADDED	Sum over all revisions of the lines of code added to a file
MAX_LOC_ADDED	Maximum number of lines of code added for all revisions
AVE_ LOC_ADDED	Average lines of code added per revision
LOC_DELETED	Sum over all revisions of the lines of code deleted from a file
MAX_LOC_DELETED	Maximum number of lines of code deleted for all revisions
AVE_LOC_DELETED	Average lines of code deleted per revision
CODECHURN	Sum of (added lines of code – deleted lines of code) over all revisions
MAX_CODECHURN	Maximum CODECHURN for all revisions
AVE_CODECHURN	Average CODECHURN per revision
MAX_CHANGESET	Maximum number of files committed together to the repository
AVE_CHANGESET	Average number of files committed together to the repository
AGE	Age of a file in weeks (counting backwards from a specific release)
WEIGHTED_AGE	See equation (1)

Table 23. List of Change metrics used in the study [110].

In particular, Moser et *al*. introduced a new change metric, namely the number of times a file has been refactored. Moreover, they defined the change set metrics, i.e., MAX/AVE_CHANGESET in ***Table 23***, as follows: the change set of a file *x* is the number of files that have been committed together with file *x* (within a time frame of 2 minutes) to the repository (it is similar to the notion of co-changes [113]). Finally, they compute the AGE of a file in weeks, starting from the release date and going back to its first appearance in the code repository. They define WEIGHTED_AGE as follows:

$$Weighted\ Age = \frac{\sum_{i=1}^{N} Age(i) \times LOC\_ADDED(i)}{\sum_{i=1}^{N} LOC\_ADDED(i)} \tag{1}$$

where: (1) *Age(i)* is the number of weeks starting from the release date for revision *i* and (ii) *LOC_ADDED(i)* is the number of lines of code added at revision *i*. In particular, WEIGHTED_AGE takes into account that defect proneness not only depends on the size of a file's changes but also when such changes occurred [112].

As remarked in [110], these set of change metrics is obviously only one possible proposal for change metrics we can extract from a CVS repository. Slightly different metrics have been proposed such as month with most revisions, average days between revisions, or relative measures for code added/deleted, and other [114].[11]

Previous Defects. Research effort has been devoted for analyzing the predictive power of past defects. One of the most known work is the approach of Kim at *al*. [81]. Kim at *al*. analyzed the version history of 7 open source software systems to predict the most fault prone entities and files. They assume that faults do not occur in isolation, but rather in bursts of several related faults. ***Table 24*** lists the different kinds of locality, which Kim at *al*. believe for bug occurrences.

Bug locality	Description
Changed-entity	If an entity was changed recently, it will tend to introduce faults soon.
New-entity	If an entity has been added recently, it will tend to introduce faults soon.
Temporal	If an entity introduced a fault recently, it will tend to introduce other faults soon.
Spatial	If an entity introduced a fault recently, "nearby" entities (in the sense of logical coupling) will also tend to introduce faults soon.

Table 24. Different kinds of bug locality considered in [81].

The Kim at *al*. prediction algorithm is executed over the change history of a software project. In particular, the algorithm yields a small subset (usually 10%) of the projects files or functions/methods that are most fault-prone. It is basically based on the cache mechanism for holding the current list of the most fault-prone entities. Therefore, locations that are likely to have faults are cached. Specifically, starting from the location of a known (fixed) fault, the algorithm caches: (i) the location itself, (ii) any locations changed together with the fault, (iii) recently added locations, and (iv) recently changed locations.

In [81], the *BugCache* algorithm and the *FixCache* algorithm for maintaining a cache based on fault localities are described and evaluated. *BugCache* updates the cache at the moment a fault is missed, that is, not found in the cache. The evaluation of this algorithm provides empirical evidence that fault localities actually exist. On the contrary, the *FixCache* algorithm shows how to turn localities into a practical fault prediction model. Kim at *al*. showed that *FixCache* predicts further faults with high accuracy. In particular, they have pointed out that combining a cache model with different heuristics for fault prediction, the *FixCache* algorithm has an accuracy of 73%-95% using files and 46%-72% using methods/functions. Details on these algorithms can be found in [81].

[11] Other examples of metrics can be found in [110].

Complexity (entropy) of code changes. As remarked [115], a wealth of metrics, which measure the complexity of the source code, have been introduced in literature. On the contrary, Hassan in [115] has focused on the complexity of the code change process, i.e., the pattern of source code modifications. He proposed to quantify the complexity over time by using historical code changes instead of source code attributes.

Source code modifications are done by developers to implement new features and repair faults. In particular, three types of modifications are identified: (i) the Fault Repairing modifications (FR) which are done to fix a fault; (ii) general maintenance modifications which are mainly bookkeeping modifications; and (iii) Feature Introduction modifications (FI) which add or enhance features. The Hassan approach uses FIs to calculate the complexity of the code change process. In contrast, FRs are not used in calculating the complexity of the change process, but are used for validating the results in the paper case study.

Hassan conjectured that: "A complex code change process negatively affects its product, the software system. The more complex changes to a file, the higher the chance the file will contain faults."[115]. The approach basically predicts defects using the entropy (or complexity) of code changes. In particular, the approach exploits the Shannon Entropy in order to measure, over a time interval, how distributed changes are in a system. The approach has been validated empirically through a case study on six large open source projects. The case study results demonstrated that the number of prior faults is a better predictor of future faults in comparison to the number of prior modifications. Moreover, the author also figured out that predictors based on our change complexity models are better predictors of future faults in large software systems (in contrast to using prior modifications or prior faults).

Churn of Source Code Metrics. D'Ambros et al. in [107] proposed to use churn of source code metrics to predict post release defects. The rationale behind this direction is that higher-level metrics may better model code churn than simple metrics like addition and deletion of lines of code. They sample the history of the source code every two weeks, and compute the deltas of source code metrics for each consecutive pair of samples.

Entropy of Source Code Metrics. D'Ambros et al. in [107] also extended the concept of code change entropy [115] to source code metrics (i.e., the listed in ***Table 17***). Specifically, they measured the complexity of the variants of a metric over subsequent sample versions. The entropy is minimum, for example, if in the system the WMC changed by 100, and only one class is involved. In the contrast, the entropy is higher, for example, if 10 classes are involved with a local change of 10 WMC. More details on this approach can be found in [107].

9 SUMMARY OF INDICATORS FOR DECISIONS MODELS

In this section, we present the overall conclusions of our work in the context of findings expected and novelty of our contribution. To the best of our knowledge, this is the first attempt to combine cost, schedule/time, and software quality and project management and application level coordination, for developing, and managing software applications. We believe that optimizing design and management of next generation systems can only be handled effectively by modelling and exploiting an explicit coordination between the, usually conflicting, quality decisions and the project management decisions (i.e., schedule/time and cost-related decisions).

We also envision that our approach will assist software designers/maintainers and software project managers during the whole software system lifecycle. Therefore, we believe that SBSE methodologies combined with multi-objective optimization (and other existing decision-making methods) and software quality validation techniques, we will address the major challenges, which now are claimed for next generation software systems (e.g., the ones for (self-) adaptive systems) and for search-based project management. Specifically, these are high level research goals (i.e., long term objectives) that we intend to achieve. We will refine these high level goals into more concrete subgoals (i.e., short term objective) until it is possible to objectively measure their satisfaction.

We claim that addressing the highlighted challenges will require the contributions from academia and industrial experts in different fields including not only search-based optimization and quality/cost/time assessments but also the experimentation of our approach on real world case studies by considering realistic model parameter values, as well as integration of our frameworks.

To address this latter point, we plan to analyze effort and time necessary to incorporate our solutions into real-world systems: as intended in the plan of Iceberg project, this is going to be addressed with some industrial scenarios provided by industrial partners, namely Assioma.net and DEISER. This part is the result of task 2.5 of the Iceberg project and it is included in this document in Section 10.

As remarked in Section 7, we intend to conduct an interview-survey in which several representatives of industry will be involved. This information is essential to validate which options from SOTA (State Of The Art) could be feasible for practitioners according to SOPA (State Of The Practice). We plan to conduct our survey in multiple stages. We are currently working on *Questionnaire Refinement* (see Step 3 of Table 5). As discussed in Section 7.4, we have already discussed with some industry representatives our questionnaire.

Regarding these fist questionnaire's results, we can remark the following points, which we intend to exploit for the next steps of our interview survey (see Table 5). Practitioners usually deal with few software metrics, or defect (cost, schedule, and time) data (see Table 25, Table 26, Table 27, Table 28).

Type	Characteristic	Metric Name	Description
Process	Schedule	WEEK	Calendar week
Process	Schedule	DAYS	Calendar Day
Process	Schedule	HOURS	Time in hours to develop/maintain the software system.

Table 25: Data Collection Unit per Schedule dimension

Type	Characteristic	Metric Name	Description
Product	Quality	INCIDENT	*Incident Report:* Identification of the associated incident if the failure report was precipitated by a service desk or help desk call/contact.
Product	Quality	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.

Table 26: Data Collection Unit per Quality dimension

Type	Characteristic	Metric	Description
Product	Size	MB	Megabyte
Product	Size	FP	Function Point
Product	Size	LOC	Number of lines of code

Table 27: Data Collection Unit per Size dimension

Type	Characteristic	Metric name	Description
Resources	Effort	PERSON-HOUR	Cost per hour to develop/maintain the software system.
Resources	Effort	PERSON-DAYS	Cost per day to develop/maintain the software system.
Resources	Cost	MONEY	Money value (per hour/day/week/month) average or differentiated by employee.

Table 28: Data Collection Unit per Cost dimension

Therefore, we will investigate the effort required for collecting additional data (e.g., additional software metrics or particular cost factors, like the one for test cases generations or execution). We will better investigate how source code metrics used in the SOTA (see ***Table 29***) are collected in the industry, and which is the effort required for collecting it. Since we have pointed out that practitioners already use some tools for source code metrics evaluation (e.g., Sonar), and bug tracking (e.g., JIRA), or they are willing to adopt them in their company, we believe that these kinds of software metrics could be easy collected and evaluated. We will analyze which is the effort required for using in industry the tools for software metrics evaluation (see ***Table 30***). As a consequence, we will investigate the "feasibility" of existing prediction approaches based on these source code metrics. Besides using these kinds of metrics, we will also exploit the Change Metrics (CM) potential (see ***Table 31***). In fact, we also believe in the practical application of these metrics.

Type	Characteristic	Metric name	Definition
Product	Structure	WMC	Weighted Method Count
Product	Structure	DIT	Depth of Inheritance Tree
Product	Structure	RFC	Response For Class
Product	Structure	NOC	Number Of Children
Product	Structure	CBO	Coupling Between Objects
Product	Structure	LCOM	Lack of Cohesion in Methods
Product	Structure	FAN_IN	Number of other classes that reference the class

Product	Structure	FAN_OUT	Number of other classes referenced by the class
Product	Structure	NOA	Number of attributes
Product	Structure	NOPA	Number of public attributes
Product	Structure	NOPRA	Number of private attributes
Product	Structure	NOAI	Number of attributes inherited
Product	Size	LOC	Number of lines of code
Product	Structure	NOM	Number of methods
Product	Structure	NOPM	Number of public methods
Product	Structure	NOPRM	Number of private methods
Product	Structure	NOMI	Number of methods inherited
Product	Structure	AHF	Attribute Hiding Factor
Product	Structure	MIF	Method Inheritance Factor
Product	Structure	AIF	Attribute Inheritance Factor
Product	Structure	MHF	Method Hiding Factor
Product	Structure	POF	Polymorphism Factor
Product	Structure	COF	Coupling Factor
Product	Structure	SIX	Specialisation Index per Class

Table 29: Source code metrics

Tools	
Jdepend	Xradar
JCSC	Checkstyle
QALab	Sonar
CKJM	Classycle
CKJM extended	Jlint
Panopticode	Sonar Plugins
Same	Squale
FindBugs	G++
JavaNCSS	Columbus
PMD/CPD	Moose

Table 30: Tools for software metrics evaluations

Type	Characteristic	Metric name	Definition
Process	Frequency	REVISIONS	Number of revisions of a file
Process	Frequency	REFACTORINGS	Number of times a file has been refactored
Process	Frequency	BUGFIXES	Number of times a file was involved in bug-fixing
Process	Size	AUTHORS	Number of distinct authors that checked a file into the repository
Process	Size	LOC_ADDED	Sum over all revisions of the lines of code added to a file
Process	Size	MAX_LOC_ADDED	Maximum number of lines of code added for all revisions
Process	Size	AVE_ LOC_ADDED	Average lines of code added per revision
Process	Size	LOC_DELETED	Sum over all revisions of the lines of code deleted from a file
Process	Size	MAX_LOC_DELETED	Maximum number of lines of code deleted for all revisions
Process	Size	AVE_LOC_DELETED	Average lines of code deleted per revision
Process	Size	CODECHURN	Sum of (added lines of code – deleted lines of code) over all revisions
Process	Size	MAX_CODECHURN	Maximum CODECHURN for all revisions
Process	Size	AVE_CODECHURN	Average CODECHURN per revision
Process	Size	MAX_CHANGESET	Maximum number of files committed together to the repository
Process	Size	AVE_CHANGESET	Average number of files committed together to the repository
Process	Size	AGE	Age of a file in weeks (counting backwards from a specific release)
Process	Size	WEIGHTED_AGE	See equation (1)

Table 31. List of Change metrics used in the study [110].

Finally, we have realized that the testing is a typical activity in the industry, and practitioners are willing to invest to improve it. We think that testing is also a good "provider" of data for the cost, schedule and time indicators. Therefore, we will investigate, for example, (i) which are the main features of these tools adopted for the testing, (ii) how these tools could be integrated with other tools (e.g., the ones for cost or time evaluations) in order to obtain both cost and time data (e.g., like the one for test cases generations or execution).

Table 32 summarizes the feasibility of data gathering.

Type	Characteristic	Metric	Feasibility
Resources	Effort	All	High
Product	Size	All	High
Resources	Cost	MONEY	High
Product	Quality	Incidents and Defects	High
Product	Quality	Source Code Metrics related to Structure	High
Product	Quality	Source Code Metrics related to Size	High
Process	Quality	Change Metrics related to Frequency	Good
Process	Quality	Change Metrics related to size	To be investigated

Table 32. Feasibility Analysis of indicators for decisions models

10 EVALUATION SCENARIOS

In this section, we describe the validation scenarios that are used in the ICEBERG project. The scenarios are described adhering to the common methodology, criteria, metrics, process, and models. The scenario description is composed from the quality activities process, the scenario objectives and the test data.

10.1 METHODOLOGY

The software development process is a structure imposed on the development of the software product containing information regarding the steps taken in project implementation. In the ICEBERG project, the following process is used:

- ***Process of requirements management***: this process is related to analysis phase;
- ***Process of development***: this process is related to design phase and implementation phase;
- ***Process of functional test***: this process is related to quality assurance phase on verification and validation of functional requirements;
- ***Process of non functional test***: this process is related to quality assurance phase on verification and validation of non functional requirements (maintainability, usability, performance, security, etc.);
- ***Process of change management***: this process is related to deployment phase;
- ***Process of production management***: this process is related to post production phase.

10.1.1 ITIL Methodology

ITIL (Information Technology Infrastructure Library - http://www.itil-officialsite.com/), is a set of practices for IT service management that focuses on aligning IT services with the needs of business. Service desk is one of the four ITIL functions and is associated with the service operation lifecycle stage. Main task includes handling incidents and requests.

End user requests (EUR) are collected using a specific service desk management system. Each “ticket” (row in data set) can be an **incident** or a **service request**.

A **service request** is a normal request related to the management of the service.

An **incident** is an event which is not part of the standard operation of a service and which causes or may cause disruption to or a reduction in the quality of services and customer productivity.

Incident could be classified based of root cause of the problem in:

- **Known error**: condition identified by successful diagnosis of the root cause of a problem, and the subsequent development of a work-around.
- **Problem**: a condition often identified as a result of multiple incidents that exhibit common symptoms. Problems can also be identified from a single significant incident, indicative of a single error, for which the cause is unknown, but for which the impact is significant.

10.2 Criteria

The criteria descriptions are structured in a general way emphasizing their reusability in various evaluation scenarios. The criteria cover typical processes of the software life cycle. Each process of the software lifecycle allows specific phase information acquisition. The standard information collection includes metrics and data describing **time**, **cost** and **quality** criteria. The additional criteria of **maturity** and **diffusion** of each process in scenarios described in this deliverable.

Time is the ability to perform actions fast. **Cost** quantifies the amount of resources assigned to the task under consideration. **Quality** quantifies whether the results coincide with desired outcomes. **Maturity** describes how well a process is established in the enterprise setting. **Diffusion** describes how often a process is used within the company.

Time is associated with software development process temporal constraints. Those temporal constraints can influence on both software development and quality assurance activities. For instance, project milestones and average time for end user request resolution are used for estimating time.

Cost is associated with software development process resource constraints. Those resource constraints influence on any step on the software development process. For instance, amount of human effort and computational resources are used for estimating cost.

Quality is associated with constraints introduced with current requirements and development process chosen for software development. Those constraints influence on software development activities laying foundation for the software acceptance phase. For instance, functional points derived from the requirements specification and code quality metrics are used for estimating quality.

Maturity is associated with constraints of the chosen software development and/or quality assessment process. Those constraints influence on the ability to perform software development process activities in effective and efficient manner. For instance, planning phase factors and development phase factors are used for estimating maturity.

Diffusion is associated with adoption constraints of the modern software development processes. Those constraints influence on the ability of the organization to use current software development process in effective manner. For instance, change, configuration and release phase factors are used for estimating diffusion.

An example maturity and diffusion evaluation is given on ***Table 33*** illustrating the metrics associated with the criteria.

	Maturity				Diffusion			
Process factors	**None**	**Low**	**Medium**	**High**	**None**	**Low**	**Medium**	**High**
Process of requirements management								
Process of development (use of repository, standard guidelines, reference application stack, …)								

Process of functional test (structured bug collection?, type of functional test, ...)								
Process of non functional test (type of non functional test, static code analysys, performance, security, ...)								
Process of change management (change, configuration & release phase factors)								
Process of post production phase (ITIL, application & system operation phase factors)								

Table 33: Maturity and diffusion evaluation

10.3 Metrics

The metrics that are used for criteria evaluation include:

- Requirements phase metrics and data influencing the process of requirements management, are for example, and function point for functional requirements.
- Development phase metrics and data influencing the process of development are for example, the number of developers involved, the estimated man days for development phase.
- Functional quality metrics and data influencing the process of functional testing, are for example, number of bugs, structured bug collection, structured test cases, bug history report, the number of testers involved, the estimated man days for test phase.
- Non functional quality metrics and data influencing the processes of non functional testing, are for example, software metrics (lines of code, cyclomatic complexity, code coverage, code churn), data extracted during performance test, data retrieved from security test.
- Change configuration phase metrics and data are for example the number of standard changes, the number of emergency changes, and the frequency of releases of a specific component.
- Post production metrics and data may be collected if there is a service desk group. Typical metrics and data are the number of tickets, incidents are collected according to ITIL methodology (see Section 9.1.1).

The metrics used in **time** evaluation include estimated man days for development phase, estimated man days for test phase, frequency of releases of a specific component, and time in days for defect resolution.

The metrics used in **cost** evaluation include number of developers involved, number of testers involved, and cost of non quality (cost of bug resolution + cost of the incident management).

The metrics used in **quality** evaluation include function points, number of bugs, structured bug collection, structured test cases, bug history report, software quality metrics, data extracted during performance test, data retrieved from security test, the number of standard changes, the number of emergency changes, the number of tickets, incidents, defects with status other than closed, opened and resolved in production and testing environment, defects with status other than closed stratified by their severity, average number of service requests, number of service requests, impact of bug on the number of incidents, impact of bug on the number of incidents opened within subsidiaries, and number of branches.

Structured bug collection, structured test cases, bug history report, data extracted during performance test, and data retrieved from security test metrics are estimated using boolean values. *True* value stands for presence, *false* for absence of the given feature.

Processes described in Section 10.1 are assessed in accordance to maturity and diffusion criteria. Each of these criteria is associated with a single metric. The values of the metrics are

- None;
- Low;
- Medium;
- High

Maturity estimation is performed by a human expert. Diffusion estimation is based on the level of diffusion of the factors under consideration in the organization.

In **maturity** evaluation *none* is assigned if the process is not defined; *low* is assigned if the process is defined but is not effective; *medium* is assigned if the process is defined and effective but need integration with other process; *high* is assigned if the process is defined and effective taking into the account that it is integrated with other process.

In **diffusion** evaluation *none* is assigned if the process is not used; *low* is assigned if the process is used by less than 20% of projects; *medium* is assigned if the process is used by less than 60% of projects; *high* is assigned if the process is used by at least 60% of projects.

10.4 Process

The data flow of the evaluation process used in the project is depicted on Figure 4. The training data is loaded into machine learning component along with prediction models. The output of the machine learning component is the learned model. Prediction component uses this model with input data to predict the metric values for the given dataset.

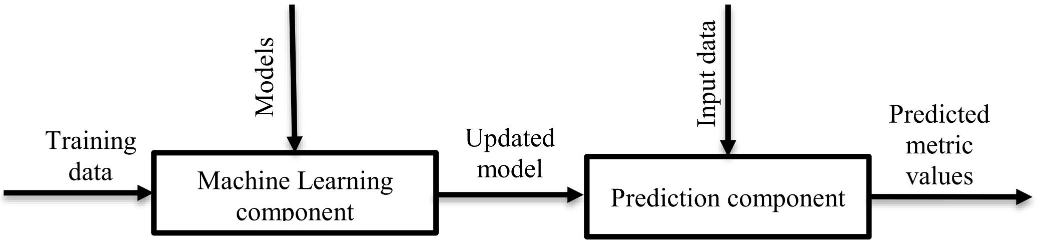

Figure 4. Evaluation process

The testing dataset is split into training data and input data. The optimal size of the training data depends on the model used.

10.5 Scenarios

Scenarios are relevant to the topic of the ICEBERG project because they deliver the data allowing to establish the relationships among criteria using variety of metrics available for computation. Scenarios are adaptable due to the presence and availability of data allowing their analysis. Scenarios are scalable because they contain large amount of data allowing discovery of the relationships among the criteria under consideration. Scenarios are extendable because of the approach tested in them, for instance, in computation metrics for various criteria, can be used in other sectors.

10.5.1 Medical Company

This scenario describes application of the QA process at the Italian company that deals with the diagnosis, application and commercialization of technical medical solutions. The software described in this section was used at the company branches. In this scenario, data was collected within 16 months. Data is related to test phase and post production phase (data provided from service desk).

10.5.1.1 Process

In the reality, the test process is not perfect and immediate, but it follows the workflow depicted in Figure 5 for the system under analysis.

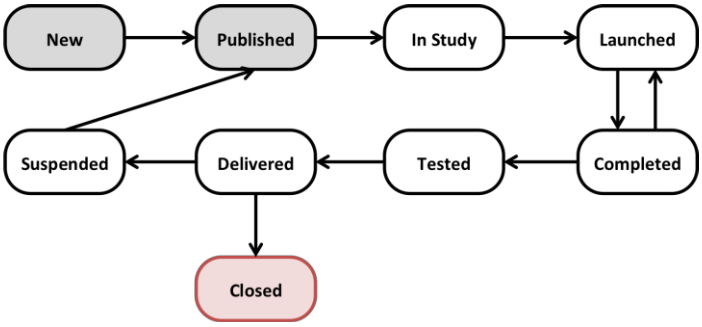

Figure 5: Workflow of the test process. Statuses in gray represent the "idle" part of the process, where issues are queued waiting to be processed.

When an issue is opened, it becomes *new* and it is queued, waiting to be processed (*published* status). Once an issue starts to be processed (*in study*), it is assigned (*launched*) to a developer and, once *completed*, it could be assigned to another developer for further processing, when needed. Then, the amendment is *tested*, *delivered*, and finally *closed*. It may happen that the testing process may fail. In this case, the issue becomes *suspended* after delivered, and then reopened again for another cycle of processing (transition in the *published* status). From the data, we also found issues that never go in the closed status, either because still under processing (e.g., this happens for recent issues opened in January 2014) or because finally classified with a "won't fix" resolution.

10.5.1.2 Scenario Objective

The objective of this scenario is to understand if it is possible predict the number of tickets during post-delivery phase, using data related to previous phases (especially test phase). Not all tickets open from service desk are related with software problem (using ITIL terminology not all tickets are incident, could be also service request). We would understand if it is possible predict the number of tickets based on data concerning test phase. Achieving this result will assist in branch budget and staffing decisions. This will also give the opportunity to estimate the optimal release time for the given residual defectiveness level.

10.5.1.3 Test Data

The test data for evaluation maturity and diffusion is depicted in ***Table 34***.

	Maturity				Diffusion			
Process factors	**None**	**Low**	**Medium**	**High**	**None**	**Low**	**Medium**	**High**
Process of requirements management			X				X	
Process of development (use of repository, standard guidelines, reference application stack, ...)								
Process of functional test (structured bug collection, type of functional test, ...)			X			X		
Process of non functional test (type of non functional test, static code analysys, performance, security, ...)		X				X		
Process of change management (change, configuration & release phase factors)								
Process of post production phase (ITIL, application & system operation phase factors)				X			X	

Table 34: Medical company scenario: maturity and diffusion

The test data for evaluating time, quality and cost is a snapshot of the bug tracking database, of test management system and of the service desk environment, that describe the errors encountered during software testing phase along with communications among users of the quality assurance process, and software engineers. In this section, data related to this process will be described:

1) Process of functional test:
    a. Change request report
    b. Bug report
    c. Bug history report
    d. Test report
2) Process of post-production phase
    a. End user request (ticket)

10.5.1.3.1 Change Request Report

The ***change request report*** provides information related to changes to be made to functionality of the system. The change requests are numbered and assigned a unique key prefixed with application name. The change requests are associated with reporter and assignee who are opening and closing the bugs. Status fields include severity, status, resolution, customer reference and labels with additional information. Status field can take cancelled, closed, completed, delivered, in study, launched, new, published, quoted, suspended, and tested values. The change request information is enriched with the dates of change request creation and update along with the versions where change request was discovered and resolved. Resolution field can take duplicate, fixed, incomplete, invalidated, not appropriate, out of perimeter, unresolved, and won't fix values. It's composed from attributes listed in ***Table 35***.

Attribute name	Attribute description	Example(s)
Number	number of changes in the change request status	1,2,3,…
Key	key for the tuples representing a change in the change request status	CR-1, CR-2, CR-3,…,
Summary	short description of the change request	new product advanced search
Description	complete description of change request, can be empty	clicking on a chart it is displayed full screen
Reporter	first and last name of the reporter	
Assignee	first and last name of the assignee, can be empty	
Status	status of the change request	published, delivered, in study,…
Created	date and time of the change request modification request creation	Wednesday 03 October 2012 14:47
Component/s	components involved in the change request resolution, can be empty	Frontend
Subcomponents	subcomponents involved in the change request resolution, can be empty	script
Updated	date and time of the change request status update	Wednesday 27 March 2013 16:18
Affects Version/s	affected version number, can be empty	1.4.1, 1.4.4, 1.5.2,…
Fix Version/s	fixed version number, can be empty	1.2.1
Resolution	whether a change request was resolved	unresolved, fixed,…
Customer reference	who opened a change request, can be empty	obtained by sales on October 9th
Labels	additional comments, can be empty	release CR2 2nd phase 15/06/2013

Table 35: The change request report

10.5.1.3.2 Bug Report

The ***bug report*** part describes the information related to bugs discovered in the application. The bugs are numbered and assigned a unique key prefixed with application name. The bugs are associated with reporter and assignee who are opening and closing the bugs. Status fields include severity, status, resolution, customer reference and labels with additional information. Severity field value can take major, minor, and blocking values. Status field can take cancelled, closed, completed, delivered, in study, launched, new, published, suspended, tested, and waiting for information values. The bug information is enriched with the dates of bug creation and updates along with the versions where the bug was discovered and resolved. Resolution field can take cannot reproduce, duplicate, fixed, incomplete, incorrect value, invalidated, not a defect, not appropriate, out of perimeter, unresolved, and won't fix values. The bug reporting part is composed from the attributes listed in Table 36.

Attribute name	Attribute description	Example(s)
Number	number of a change in the bug status	1,2,3,…
Key	key for the tuples representing a change in the bug status	BUG-1, BUG-2, BUG-3,…,
Summary	short description of the bug	Sales - Quote - Graphic missing
Description	complete description of bug, can be empty	when I select a profile does not allow me to view the features
Reporter	first and last name of the reporter	
Assignee	first and last name of the assignee	
Severity	severity of the bug encountered, can be empty	major, minor,…,
Status	status of the bug	open, closed,…
Created	date and time of the bug modification request creation	Friday 07 September 2012 10:02
Components	components involved in the bug resolution, can be empty	Frontend
Subcomponents	subcomponents involved in the bug resolution, can be empty	sales
Updated	date and time of the bug status update	Friday 21 September 2012 15:12
Affects version/s	affected version number, can be empty	1.0.0
Fix versions/s	fixed version number, can be empty	1.2.0
Resolution	whether a bug was resolved	Fixed
Customer reference	who opened a bug, can be empty	Turin branch
Labels	additional comments, can be empty	Online/Offline

Table 36: The bug report

10.5.1.3.3 Bug History Report

The ***bug history report*** part describes attributes related to change in the bug status. It contains data about the bug, its number and identifier in addition to the data of who and when modified the bug status. It is composed from attributes listed in ***Table 37***.

Attribute name	Attribute description	Example(s)
Issue	key from the bug and customer reporting parts	BUG-4124, BUG-4123, …

#Issue	number from the bug and customer reporting parts	4124, 4123, …
Modified by	first and last name of the person modified the bug	
Modification date	bug status modification date, can be empty	20/01/2014 16:36
Field	what is changed in relation to a given bug, can be empty	labels, status, assignee, …
Original Value	original value of the changed attribute, can be empty	in study, launched, …
New Value	new value of the changed attribute, can be empty	launched, completed, …

Table 37: The bug history report

10.5.1.3.4 Test Report

The ***test report*** part relates test suites and test cases with their priority including also the versions for which a given test suite was run on a given test case. Priority field can take medium and high values. The test report also includes the information on number of tests passed, failed, not run and executed for a given test case. It is composed from attributes listed in ***Table 38***.

Attribute name	Attribute description	Example(s)
Test suite	test suite run against the given test case	phase 1/functional test/home page
Test case	test case string	TC-209: Layout, TC-221:Main menu,…
Priority	priority of the test case	Medium, High,…
Version attributes (1.0.0, 1.1.0, 1.1.1, 1.2.0, 1.2.1, 1.2.2, 1.2.3, 1.3.0, 1.3.1, 1.4.0, 1.4.1, 1.4.2, 1.4.3, 1.4.4, 1.4.5, 1.5.0, 1.5.1, 1.5.2, 1.5.3, 1.5.4, 1.6.0, 1.6.1, 1.6.2, 1.6.3, 1.6.4, 1.7.0, 1.7.1, 1.7.2, 1.8.0, 1.8.1, 1.8.2, 1.8.3, 1.8.4, 1.9.0, 1.9.1, 1.9.2, 1.9.3)	result of the test execution on the given version of the system	Passed, Blocked,…
Last build	result of the test execution on the last build of the system	Passed
Last execution	result of the test last execution	Passed
Passed	number of versions on which the test passed	1, 3, …
Failed	number of versions on which the test failed	1,2, …

Not run	number of versions on which the test was not run	35, 36, …
Executions	number of test suite executions	1,2, …

Table 38: The test report

The data populating these attributes is available for 2 test phases comprising functional, interface, cross functional, and functional non regression tests.

10.5.1.3.5 End User Request (ticket)

According to ITIL methodology, in this scenario a service desk group collects and manages ***end user requests***.

Each ticket, provide dates EUR was submitted and closed along with dates when the data was imported and analyzed, status and branch of the company from which request originates. EURs are also associated with times of the assignment and resolution in days along with reference for EUR and its quality properties of whether the request was open and closed within a day and worked as expected. They also include detailed information about the dates including weeks when they were opened and closed, days and month of submission and finalization. The part includes attributes listed in ***Table 39***.

Attribute name	Attribute description	Example(s)
Ticket number	EUR number	117, 119,…
Date submitted	date EUR submitted	04/07/2013
Date closed	date EUR closed, can be empty	04/07/2013
Status	EUR status	closed
Date footprint imported	date of the data import that must be after submission date	08/08/2013
Date of final analysis	date of final analysis, can be empty	18/07/2013
GTA	time of assignment in days, can be empty	14
GTR	time of resolution in days, can be empty	14
Supplier ticket reference	key from the bug report and change request parts, in some cases refer to problems with string and empty identifiers	BUG-2814, Updates installation
Type request	type of the request encountered, can be empty	modification query
Open within the day	request that is open within the day	1
Close within the day	request that is closed within the day	1
Worked as expected	software worked as expected	1
Week of submission	week of the year on which the software modification request was obtained	27
Week of resolution	week of the year on which the software modification request was resolved	35
Day of submission	day of the month submitted	4, 5, …

Month of submission	month of the year submitted	7
Day finalized	day of the month the work was finalized	7
Month finalized	month of the year the work was finalized	7
Branch	branch of the company from which the request is originating	Turin
First Name	first name of the person to whom the bug is assigned, can be empty	Sara

Table 39: The end user requests

10.5.2 Telecommunications Company

An Italian telecommunications company provides landline, broadband Internet, and digital TV services. The company employs huge number of IT personnel. It employs both internal and external software and test factories. In this scenario, data was collected within 12 months.

10.5.2.1 Process

The software testing phase includes functional, integration and production phases. The test process is depicted on Figure 6.

When an issue is opened (*1-Open*) it can either be assigned irrelevant (*6-Obsolete*) status or proposed for rejection (*5a-Rej. Proposed*) and rejected (*5-Rejected*). The third option includes acceptance (*1a-Accepted*) and delivery for workflow draw (*3a-Delivered to FW*). From this status, issue might become either fixed (*7-Closed*) or not fixed (*2-Reopen*). The third option includes fixing (*3-Fixed*) status. After that the issue might be either fixed (*7-Closed*) or not fixed (*2-Reopen*). The third option includes readiness to test (*12-Ready to test*). The issue with this status can be considered either as fixed (*7-Closed*) or not fixed (*2-Reopen*). The third option includes testing (*4-Tested*) and release (*11-Release*) after that the issue is considered as fixed (*7-Closed*). The not fixed status (*2-Reopen*) leads to either return to bug acceptance (*1a-Accepted*) or bug rejection proposal (*5a-Rej. Proposed*).

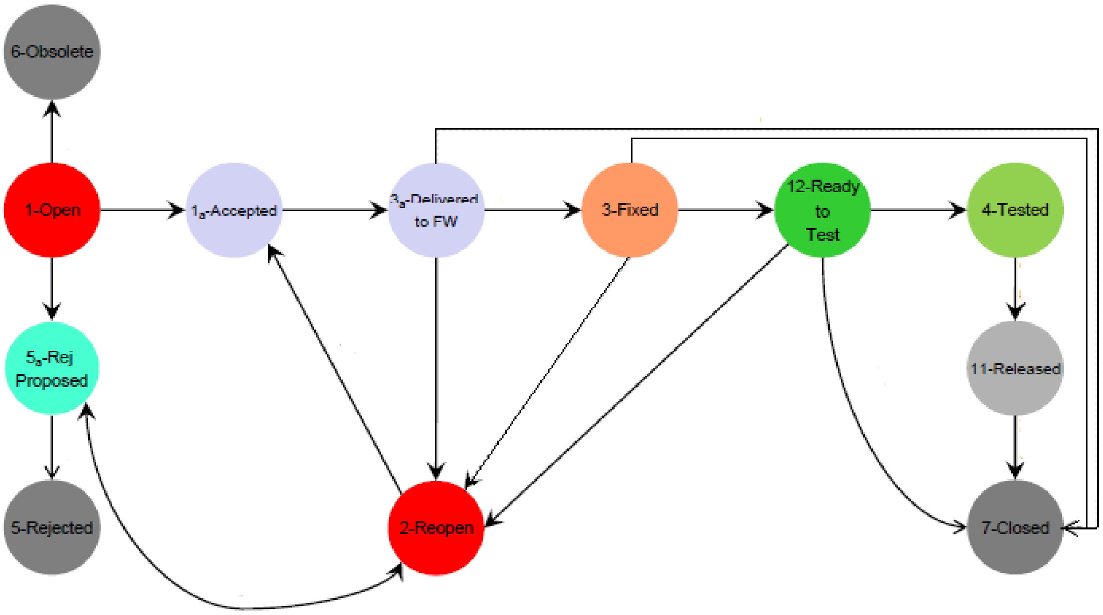

Figure 6: Test process.

10.5.2.2 Scenario Objective

The main objective of the scenario is to define what correlations exist between the incident detected in production environment and the processes of software quality. The current process of quality assurance includes integration testing, acceptance testing, deployment and configuration management, and performance testing. The process of testing is structured; however, its process is defined according to start-up structure with several incidents open for years. Moreover, the testing process is poorly defined and planning of the quality related activities is chaotic.

The second objective of this scenario is to predict the number of change requests on various phases of testing, using the testing data of the previous phases. Not all change requests are related with software problem (using ITIL terminology not all change requests are incident, could be also service request). The aim is to understand if it's possible to predict the number of change requests based on data concerning various test phases.

Achieving this results will assist in taking decisions from the actors involved in Software Quality Process. This will also give the opportunity to estimate the optimal release time for the given residual defectiveness level

10.5.2.3 Test Data

The test data for evaluation maturity and diffusion is depicted in ***Table 40***.

	Maturity				Diffusion			
Process factors	**None**	**Low**	**Medium**	**High**	**None**	**Low**	**Medium**	**High**
Process of requirements management								

Process of development (use of repository, standard guidelines, reference application stack, ...)								
Process of functional test (structured bug collection?, type of functional test, ...)			X				X	
Process of non functional test (type of non functional test, static code analysis, performance, security, ...)								
Process of change management (change, configuration & release phase factors)								
Process of post-production phase (ITIL, application & system operation phase factors)				X			X	

Table 40: Telecommunication company scenario: maturity and diffusion

The test data for evaluating time, quality and cost is a snapshot of the bug tracking database, of test management system, that describe the errors encountered during software testing phases. In this section, the data related to this process is described.

10.5.2.3.1 Change Request Report

The ***change request report*** provides information related to requested changes of the system functionality. The change requests collected on these phases are associated with bug id, project, attributes of status, severity, priority, type, phase, detection and closing dates. Severity attribute values include 1-low, 2-medium, 3-high, 4-very high, and 5-blocking. Priority attribute values include 1-low, 2-medium, 3-high, 4-very high, and 5-urgent. Type attribute values include functional bug, integration bug, delivery/installation bug, new requirements enhancement, functional analysis bug, documentation bug, performance bug, functional analysis enhancement, external system, cleaning, cleaning other system, bug out of warranty, platform bug, enhancement analysis requirements errata, enhancement infrastructural operations, and workaround. Found in phase attribute values include 01-functional, 02-integration, and 03-production. Closing date attribute is provided for closed value of the status attribute. It is composed from attributes listed in ***Table 41***.

Attribute name	Attribute description	Example(s)
Change request reference	project initiative	P130503-may advertisements HP
Bug id	key for the change request	55023
Project	application component related to the change request	RIC
Status	status of the change request, number 1 to 12 in some cases with additional letter and description	3-Fixed
Severity	severity of the change request in [1,5] interval with description	3-High
Priority	priority of the change request in [1,5] interval with description	3-High
Type	type of the change request	Request for functionality enhancement

Found in phase	phase of the change request discovery in interval [1,3] with description	03-Production
Detected on date	date of the change request discovery	03/06/2013
Closing date	subcomponents involved in the change request resolution, can be empty	09/09/2013 13:00:15

Table 41: The change request report

10.5.2.3.2 Test Report

Test report part relates change requests with defect and test numbers. It is composed from attributes listed in ***Table 42***.

Attribute name	Attribute description	Example(s)
Label	change request	P130733-TOOL Regulatory Cost Control – Reprising and Decision Management - Additional,…
Defects count	defects found	385,…
Tests count	tests performed, can be empty	1217,…

Table 42: The test report

10.5.3 Financial Company

The multinational company provides banking and financial services. Its primary businesses are retail banking, direct banking, commercial banking, investment banking, asset management, and insurance services. In this scenario, data was collected within 7 months.

10.5.3.1 Process

The data presented in this scenario corresponds to software verification process and process of the service desk. Software defects found on software verification process stage are associated with their statuses including assigned, closed, on hold, production issue, ready for retest, rejected, reopen, testing issue, and under development. The test process is depicted on Figure 7.

The work (*Under dev*.) on the open issue (*Assigned*) either leads to its reopening (*Rejected, Reopen, Under dev*.) or fixing (*Ready for migration, Ready for retest, Under retest*). Afterwards the issue either thought of as not fixed (*Reopen*) or fixed (*Closed*). *Rejected* and *Closed* statuses correspond to *production issues* while the rest correspond to *testing issues*.

Program incidents found on service desk process stage are associated with their statuses including analysis, open, closed, non-testable, in development, and on hold.

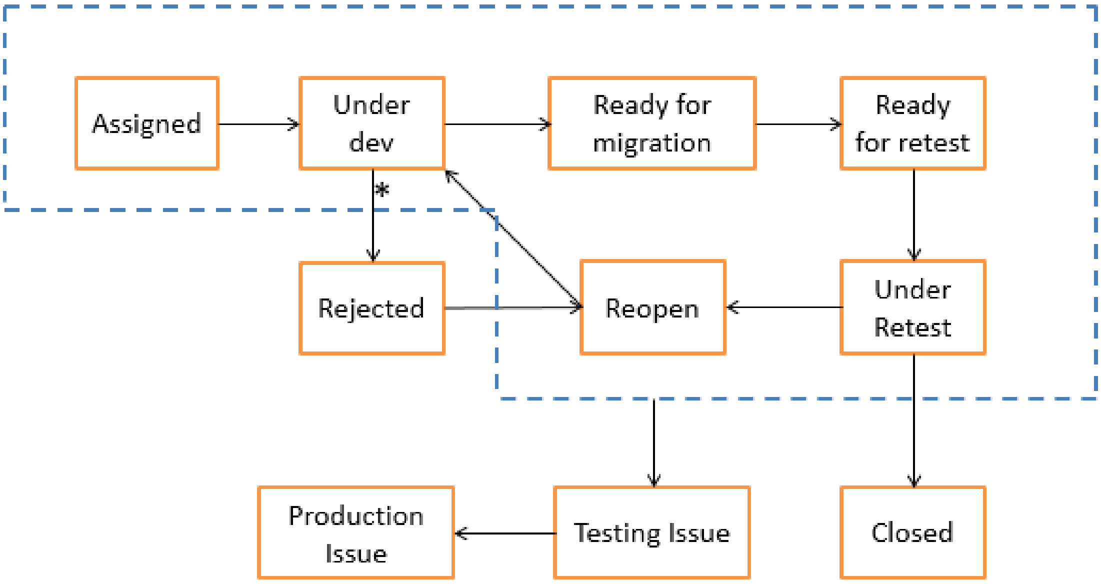

Figure 7: Test process.

10.5.3.2 SCENARIO OBJECTIVE

The main objective of the scenario is to define what correlations exist between the incidents detected in service desk process and software defects encountered in software verification process.

Achieving this results will assist in taking IT decisions. This will also give the opportunity to estimate the optimal release time for the given residual defectiveness level

10.5.3.3 TEST DATA

The test data for evaluation maturity and diffusion is depicted in ***Table 43***.

	Maturity				Diffusion			
Process factors	**None**	**Low**	**Medium**	**High**	**None**	**Low**	**Medium**	**High**
Process of requirements management								
Process of development (use of repository, standard guidelines, reference application stack, …)								
Process of functional test (structured bug collection?, type of functional test, …)			X				X	
Process of non functional test (type of non functional test, static code analysis, performance, security, …)								
Process of change management (change, configuration & release phase factors)								
Process of post-production phase (ITIL, application & system operation phase factors)				X			X	

Table 43: The Finance Scenario: Maturity and Diffusion

The test data for evaluating time, quality and cost is a snapshot of the bug tracking database, of test management system, that describe the errors encountered during software verification process. In this section, the data related to this process is described.

10.5.3.3.1 Program Defects

Program defects part connects defects ID with their summary, description along with names of who detected the defect, to whom and to which group the defect was assigned, the attributes of their severity and status, change request number, program environment descriptor, comments, modification, release and detection dates, defect priority, cause, cycle, reopening count, time to fix, and closing date. Severity attribute values include 1-low, 2-medium, 3-high, 4-very high, and 5-urgent. Defect manager priority attribute values include 1-production blocking, 2-medium priority, and 3-low priority. It is composed from attributes listed in ***Table 44***.

Attribute name	Attribute description	Example(s)
Defect ID	defect numeric identifier	19383
Summary	change request summary	service createDelegation
Description	change request description	The field CD_PRODOTTO always takes the value DDCO while on CSE is valued as CORE
Detected By	full name in format first_name.last_name	
Assigned To	full name in format first_name.last_name, in some cases group	
AssignedTo Group	group, can be empty	ICBPI
Severity	severity of change request in [1,5] interval	3-High
Status	status of the change request	Assigned
CR Number	change request identifier	8592 B
Environment	environment descriptor	SIT
Comments	messages related to change request, can be empty	<Name>, 03/03/2014: Fix effective
Modified	modification date (update of the bug information)	03/04/2014 09:32:39
ReleaseDate	release date	2013-11-30
Detected on date	detected on date	17/10/2013

Defect Manager Priority	priority assigned by a defect manager in [1,3] interval, can be empty	3-Low priority
Root Cause	cause of the defect, can be empty	environment defect
Detected in Cycle	components involved in the defect resolution, can be empty	AEA file
Num_Reopen	defect reopening count, can be empty	
Actual Fix Time	time to fix in days, can be empty	3
Closing date	date to close defect in test environment, can be empty	19/03/2014

Table 44: Defects

10.5.3.3.2 Production Issues

Production issues part connects process identifiers with the channel, production issue, service desk and link identifiers, owner, description and names of who detected the defect, to whom the issue was assigned, opening, RD and closure dates, state, severity, product, cause of the issue, whether the issue is a testing issue and whether it is testable, who are the first three developers assigned to resolve the issue and in which applications it can be found. It is composed from attributes listed in ***Table 45***.

Attribute name	Attribute description	Example(s)
Process	process identifier, can be empty	INQ.CA
Channel	channel, can be empty	institutional site
PI	production issue identifier	83771
SD	open ticket identifier, can be empty	SD238876
Link (CR/PI)	change request or production issue, which resulted in this defect	8592
Owner	owner full name	
Description	string with description	Balance and summary for ASP clients
User	user full name	
Analyst	analyst full name	
Open	opening date	07/04/2014 10:00:01
RD	release date, can be empty	08/04/2014
Closed	closed date, can be empty	03/04/2014
State	issue state	On hold
Severity	severity in [1,5] interval	3
Product	software product, can be empty	CROSS
Root Cause	issue cause, can be empty	technical problem
Testing issue	whether issue is a testing issue	Yes

Not testable	whether issue is testable	Yes
Developer 1	developer full name, can be empty	
Developer 2	developer full name, can be empty	
Developer 3	developer full name, can be empty	
Application 1	incident application, can be empty	SmartPayments
Application 2	incident application, can be empty	
Application 3	incident application, can be empty	
Application 4	incident application, can be empty	

Table 45: Production issues

10.5.4 Banking Company

This Spanish company provides banking and financial services. Its primary businesses are grouped into the following areas: Retail Banking, Business Banking, Private Banking, Asset Management, Capital Markets and Holdings. The data are collected for 10 months.

10.5.4.1 PROCESS

The scenario is about the requests (corrective maintenance and small improvements) made at this company. The requests are those with an effort below 10 days and have to be fixed by different suppliers (this does not mean that one CR is fixed by more than one supplier). The organization has approved a group of suppliers and the workflow is intended to manage the relationship between the outsourced companies and the banking institution.

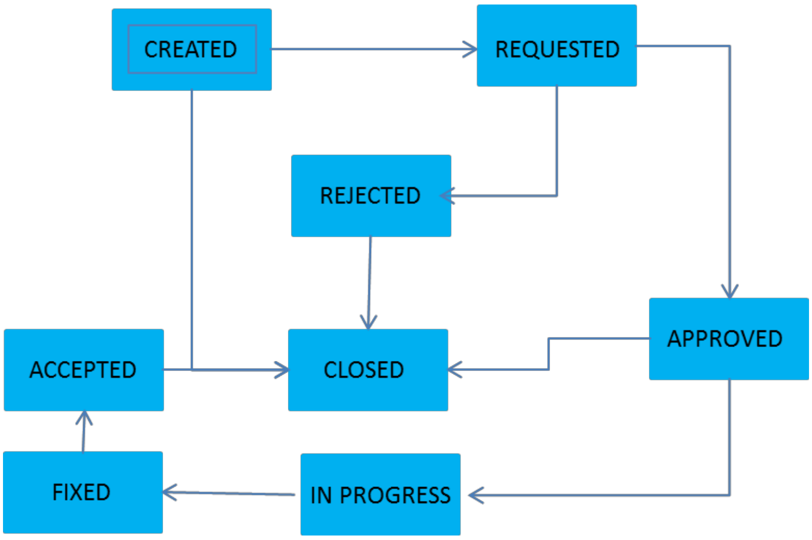

Figure 8: Test process.

The requests are created (this is a complex state where the request can be detailed. This is the part executed by the bank. Then the request is sent to the supplier (requested). The companies have to send back proposals that must be approved. Once done (the approval), the work begins (In Progress), finishes and (if OK) is closed. Of course there are transitions from different states to the closed and/or rejected status.

The system records a lot of information regarding the progress, dates, durations and so on. The idea is try to rank the work of the suppliers depending on the information submitted to the system.

The test Data Section shows the most relevant items collected (there are more data but we will focus on the fields we need to reach our goals.

10.5.4.2 SCENARIO OBJECTIVE

The main objective of the scenario is to determine if it's possible to rank the suppliers based on the collected info. One of the most critical factors for big companies to decide applying QA measures is to obtain a clear profit. This is considered a key point in the software development process so the purpose is to find a classification or rank to assess companies which is the best choice.

10.5.4.3 TEST DATA

The test data are collected using a bug tracking tool. The process collects information about Requests. These Requests are, in fact, service requests that are assigned to different suppliers.

Process factors	Maturity				Diffusion			
	None	Low	Medium	High	None	Low	Medium	High
Process of requirements management								
Process of development (use of repository, standard guidelines, reference application stack, ...)								
Process of functional test (structured bug collection?, type of functional test, ...)								
Process of non functional test (type of non functional test, static code analysis, performance, security, ...)								
Process of change management (change, configuration & release phase factors)								
Process of post-production phase (ITIL, application & system operation phase factors)			X			X		

Table 46: The Banking Scenario: Maturity and Diffusion

10.5.4.3.1 Change Request Report

The change request report includes detailed info of the requests. The information shown below is partial. There are more info but we focus on the fields that can be relevant for the purpose.

Attribute name	Attribute description	Example(s)
Change request reference	project initiative	CRBK01
Supplier	Name of the supplier	DEISER

Project	application component related to the change request	Onlinebanking
Status	status of the change request	In-Progress
Priority	priority of the change request in [1,5] interval with description	3-High
Type	type of the change request	1.SmallImpr 2.Corrective maintenance
Created	Date of the change request creation	03/06/2012
Accepted	Date when the CR was accepted. This is not the close day. It represents the start of the supplier work	03/07/2012
Finished	Date when the CR is finished	09/07/2012
Delivery	Date when the CR is delivered and deployed	10/07/2012

Table 47: The Change Request issues

10.5.4.3.2 Change Request History Report

Describes attributes related to change in the change Request. It is composed from attributes listed in ***Table 48***.

Attribute name	Attribute description	Example(s)
Change Request	key from the bug and customer reporting parts	CRBK01
Modified by	first and last name of the person modified the change request	
Modification date	bug status modification date, can be empty	20/01/2014 16:36
Field	what is changed in relation to a given bug, can be empty	Status, Priority, Type.
Original Value	original value of the changed Field	
New Value	new value of the changed Field	

Table 48: The Change Request report

10.5.5 Public Entity

The Information System Consortium of regional Public Administration is associated with management and implementation of ICT services. The entity includes over 100 of public bodies: in addition to its founders, the capital administration, all the provinces, many municipalities and associated forms, health agencies and hospitals and other agencies. The consortium works in all PA sectors: from health to production activities, from cultural heritage to administrative systems and from the territory to professional training and employment.

10.5.5.1 PROCESS

The values of the state attribute include closed, suspended, waiting, and released. The execution of the test case is assigned to specific figures. The process involves the execution of tests, starting from use cases, without a specific preparation of test cases.

10.5.5.2 SCENARIO OBJECTIVE

The main objective of the scenario is to define what correlations exist between the bugs detected in testing and post-production phases.

Achieving this results will assist in taking IT decisions. This will also give the opportunity to estimate the optimal release time for the given residual defectiveness level.

10.5.5.3 TEST DATA

The test data is collected for Java project with duration of 30 month including 5 subprojects.

In addition to information regarding testing activities the data set includes code quality metrics.

10.5.5.3.1 Requirements management

The data collected on requirements management phase includes Total Function Points (FP) along with effort estimation for phases of functional and technical analysis, development, and test. ***Table 49*** depicts the data.

Metrics	Values
Total Function Points	5.700
Functional analysis phase effort estimation	2.200 man/days
Technical analysis phase effort estimation	580 man/days
Development phase effort estimation	4.600 man/days
Testing phase effort estimation	2.900 man/days

Table 49: Requirements management metrics

10.5.5.3.2 Functional analysis

The data collected on functional analysis phase includes effort spent on use case construction including the use case number. ***Table 50*** depicts the data.

Metrics	Values
Use case construction effort	3.300 man/days
Use cases	830

Table 50: Functional analysis metrics

10.5.5.3.3 Development

The data collected on development phase includes the number of developers involved in the project, project development effort and software complexity metrics computed at the end of the project. The code quality metrics calculated for two main modules are depicted in ***Table 51***.

Metrics	Module 1	Module 2
Lines of code	161.982	275.041
Physical lines	253.376	411.931
Statements	86.468	130.748
Number of files	781	2.056
Number of classes	794	2.153
Number of packages	80	122
Number of methods	10.404	17.586

Table 51: Development metrics

10.5.5.3.4 Testing

The data collected on testing phase includes number of testers involved in the project and total testing effort. The metrics values are depicted on ***Table 52***.

Metrics	Values
Number of testers	5
Total testing effort	3.600 man/days

Table 52: Testing metrics

In addition was obtained the dataset of the bug tracking system. The attributes of the dataset include type, key, severity, state, priority, resolution, creation and update dates. The value of type attribute is “bug test”. The values for the severity attribute include blocking, critical, insignificant, and tolerable. The values of the state attribute include closed, suspended, waiting, and released. The priority attribute includes highest, high, medium, and low values. The resolution attribute includes values fixed and not resolved. The attributes are depicted on ***Table 53***.

Attribute name	Attribute description	Example(s)
Type	type of bug report	bug test
Key	key of the tuple, name of the project and number	
Severity	bug severity	insignificant
State	state of the bug resolution	released
Priority	bug priority	high
Resolution	status of the bug resolution	fixed
Created	creation date	1/10/2009 10.00

Updated	update date	1/10/2009 10.33

Table 53: Testing phase bug report

10.5.5.3.5 Post production

The attributes of bug reports include type, key, severity, state, priority, resolution, creation, update dates, and solution number. The value of type attribute is "bug test". The values for the severity attribute include blocking, critical, insignificant, and tolerable. The values of the state attribute include closed, and waiting. The priority attribute includes the following values highest, high, medium, and low. The resolution attribute includes values fixed, not resolved, duplicated, and withdrawal. The attributes are depicted on ***Table 54***.

Attribute name	Attribute description	Example(s)
Type	type of bug report	bug test
Key	key of the tuple, name of the project and number	
Severity	bug severity	insignificant
State	state of the bug resolution	released
Priority	bug priority	high
Resolution	status of the bug resolution	fixed
Created	creation date	1/10/2009 10.00
Updated	update date	1/10/2009 10.33
Solution number	number of remedy, can be empty	HD0000003633895

Table 54: Post production phase bug report

11 CONCLUSION

The quality parameters and validation scenarios were described focusing on the key factors influencing cost, time and quality. Five evaluation scenarios taken from medical, telecommunication and financial domains were described with a special emphasis put on quality parameters. The scenarios were selected based on their relevancy, adaptability, scalability and extendibility.

12 REFERENCES

[1] B. Crosby, “Quality is free. The art of making quality certain”, McGraw-Hill, 1987.

[2] H. J. Harrington, “Poor-Quality Cost: Implementing, Understanding, and Using the Cost of Poor Quality”, CRC, 1987.

[3] H. J. Harrington, “The Productivity and Quality Connection”, IEEE Journal on Selected Areas in Communication, vol. 4, nº 7, pp. 1009-1014, 1986.

[4] APM, “A History of the Association for Project Management 1972-2010”, Association for Project Management, Buckinghamshire, 2010.

[5] A. Schiffauerova y V. Thomson, “Review of Research on Cost of Quality Models and Best”, International Journal of Quality and Reliability Management, vol. 23, nº 4, 2006.

[6] S. Suthummanon y N. Sirivongpaisal, “Investigation of the Relationship between Quality and Cost of Quality in a Wholesale Company”, ASEAN Engineering Journal, vol. 1, nº 1, 2011.

[7] R. S. Pressman, “Software engineering: a practitioner’s approach (7th ed.)”, Boston: McGraw-Hill, 2010.

[8] L. M. Karg, M. Grottke y A. Beckhaus, “A systematic literature review of software quality cost research”, Journal of Systems and Software, vol. 84, nº 3, pp. 415-427, 2011.

[9] B. Boehm, “Value-Based Software Engineering”, ACM Software Engineering Notes, vol. 28 (2), pp. 1-12, 2003.

[10] J. Thorp, “The information paradox”, New York, NY: McGraw-Hill, 2003.

[11] C. Jones and O. Bonsignour, The Economics of Software Quality, Addison-Wesley Professional, 2011.

[12] D. Tajima and T. Matsubara, “Inside the Japanese Software Industry,” *Computer*, vol. 17, no. 3, pp. 34–43, Mar. 1984.

[13] J. Inglis, “Standard Software Quality Metrics,” *AT&T Tech. J.*, vol. 65, no. 2, pp. 113–118, Mar. 1986.

[14] IEEE: IEEE Std. 1044-2009. Standard Classification for Software Anomalies.

[15] http://www.best-management-practice.com/gempdf/itil_glossary_v3_1_24.pdf

[16] ISO/IEC/IEEE 24765:2010. Systems and software engineering – Vocabulary.

[17] http://standards.ieee.org/findstds/standard/610-1990.html

[18] S. Wagner, Defect Classification and Defect Types Revisited. In : Proceedings of the 2008 workshop on Defects in large software systems, NY, USA, pp.39–40 (2008).

[19] D. Vallespir, F. Grazioli,, J. Herbert. A Framework to Evaluate Defect Taxonomies. In: VI Workshop Ingeniería de Software (WIS), Argentina, pp.643-652 (2009).

[20] R. Chillarege. Orthogonal Defect Classification. In: Handbook of Software Reliability Engineering IEEE Computer Society Press and McGraw-Hill edn. M. R. Lyu (1996).

[21] B. Freimut. Developing and Using Defect Classification Schemes. IESE- Report No.072.01/E, Institut Experimentelles Software Engineering, Sauerwiesen (2001).

[22] R. Grady. Software Failure Analysis for High-Return Process Improvement Decisions. Hewlett-Packard Journal (1996).

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

[24] N. Fenton and S. L. Pfleeger, Software Metrics (2Nd Ed.): A Rigorous and Practical Approach. Boston, MA, USA: PWS Publishing Co., 1997.

[25] A. J. Albrecht and J. E. Gaffney, "Software function, source lines of code and development effort prediction: A software science validation," IEEE Trans. Software Eng, vol. SE-9, no. 6, pp. 639-647, Nov. 1983.

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

[27] E. Babbie. Survey Research Method, 2nd edition. Wadsworth Publishing Company, 1990.

[28] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. Experimentationin software engineering: an introduction. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[29] H. K. Wright, M. Kim, and D. E. Perry. Validity concerns in software engineering research. In Proceedings of the FSE/SDP workshop on Future of software engineering research, FoSER '10, pages 411–414. ACM, 2010.

[30] U. van Heesch and P. Avgeriou. Mature Architecting - A Survey about the Reasoning Process of Professional Architects. In WICSA, pages 260–269. IEEE Computer Society, 2011.

[31] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "The architecture tradeoff analysis method," in Engineering of Complex Computer Systems, 1998. ICECCS '98. Proceedings. Fourth IEEE International Conference on, Aug 1998, pp. 68–78.

[32] R. Kazman, J. Asundi, and M. Klein, "Quantifying the costs and benefits of architectural decisions," in Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on, May 2001, pp. 297–306.

[33] B. Güldali, H. Funke, S. Sauer, and G. Engels, "TORC: test plan optimization by requirements clustering," Software Quality Journal, vol. 19, no. 4, pp. 771–799, 2011.

[34] A. Bertolino, P. Inverardi, and H. Muccini, "Software architecture-based analysis and testing: a look into achievements and future challenges," Computing, vol. 95, no. 8, pp. 633–648, 2013.

[35] D. Cotroneo, R. Pietrantuono, and S. Russo, "Testing techniques selection based on ODC fault types and software metrics," Journal of Systems and Software, vol. 86, no. 6, pp. 1613 – 1637, 2013.

[36] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," IEEE Transactions on Software Engineering, vol. 37, no. 5, pp. 649–678, 2011.

[37] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya, "Software Architecture Optimization Methods: A Systematic Literature Review," IEEE Transactions on Software Engineering, vol. 39, no. 5, pp. 658–683, 2013.

[38] P. Potena, “Optimization of adaptation plans for a service-oriented architecture with cost, reliability, availability and performance tradeoff,” Journal of Systems and Software, vol. 86, no. 3, pp. 624 – 648, 2013.

[39] M. Buzdalov, A. Buzdalova, and I. Petrova, “Generation of tests for programming challenge tasks using multi-objective optimization,” in GECCO (Companion), C. Blum and E. Alba, Eds. ACM, 2013, pp. 1655–1658.

[40] G. Canfora, A. D. Lucia, M. D. Penta, R. Oliveto, A. Panichella, and S. Panichella, “Multi-objective Cross-Project Defect Prediction,”in ICST. IEEE, 2013, pp. 252–261.

[41] N. Oster and F. Saglietti, “Automatic Test Data Generation by Multiobjective Optimisation,” in SAFECOMP, ser. Lecture Notes in Computer Science, J. G´orski, Ed., vol. 4166. Springer, 2006, pp. 426–438.

[42] R. Mirandola, P. Potena, and P. Scandurra, “Adaptation space exploration for service-oriented applications,” Science of Computer Programming, vol. 80, Part B, no. 0, pp. 356 – 384, 2014.

[43] M. Jørgensen, “The influence of selection bias on effort overruns in software development projects,” Information & Software Technology, vol. 55, no. 9, pp. 1640–1650, 2013.

[44] H. K. Doloi, “Understanding stakeholders’ perspective of cost estimation in project management,” International Journal of Project Management, vol. 29, no. 5, pp. 622 – 636, 2011.

[45] C. Heimerl and R. Kolisch, “Scheduling and staffing multiple projects with a multi-skilled workforce,” OR Spectrum, vol. 32, no. 2, pp. 343– 368, 2010.

[46] H. Li and K. Womer, “Scheduling projects with multi-skilled personnel by a hybrid MILP/CP benders decomposition algorithm,” J. Scheduling, vol. 12, no. 3, pp. 281–298, 2009.

[47] M. Firat and C. A. J. Hurkens, “An improved MIP-based approach for a multi-skill workforce scheduling problem,” J. Scheduling, vol. 15, no. 3, pp. 363–380, 2012.

[48] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark, “The GISMOE Challenge: Constructing the Pareto Program Surface Using Genetic Programming to Find Better Programs (Keynote Paper),” in Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ser. ASE 2012. ACM, 2012, pp. 1–14.

[49] E. Diaz, J. Tuya, and R. Blanco, “Automated software testing using a metaheuristic technique based on Tabu search,” in Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on, Oct 2003, pp. 310–313.

[50] F. M. Kifetew, A. Panichella, A. De Lucia, R. Oliveto, and P. Tonella, “Orthogonal Exploration of the Search Space in Evolutionary Test Case Generation,” in Proceedings of the 2013 International Symposium on Software Testing and Analysis, ser. ISSTA 2013. New York, NY, USA: ACM, 2013, pp. 257–267.

[51] K. Lakhotia, M. Harman, and H. Gross, “Austin: An open source tool for search based software testing of c programs,” Information and Software Technology, vol. 55, no. 1, pp. 112 – 125, 2013, special section: Best papers from the 2nd International Symposium on Search Based Software Engineering 2010.

[52] Z. Ye, X. Zhou, and A. Bouguettaya, “Genetic Algorithm Based QoSAware Service Compositions in Cloud Computing,” in DASFAA (2), ser. Lecture Notes in Computer Science, J. X. Yu, M.-H. Kim, and R. Unland, Eds., vol. 6588. Springer, 2011, pp. 321–334.

[53] F. Rosenberg, M. Muller, P. Leitner, A. Michlmayr, A. Bouguettaya, and S. Dustdar, “Metaheuristic optimization of large-scale qos-aware service compositions,” in Services Computing (SCC), 2010 IEEE International Conference on, July 2010, pp. 97–104.

[54] S. Malek, N. Medvidovic, and M. Mikic-Rakic, “An Extensible Framework for Improving a Distributed Software System’s Deployment Architecture,” Software Engineering, IEEE Transactions on, vol. 38, no. 1, pp. 73–100, Jan 2012

[55] D. Rodriguez, M. Ruiz, J. C. Riquelme, and R. Harrison, “Multiobjective Simulation Optimisation in Software Project Management,”in Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, ser. GECCO ’11. ACM, 2011, pp. 1883–1890.

[56] F. Ferrucci, M. Harman, and F. Sarro, “Search-Based Software Project Management,” Software Project Management in a Changing World (Springer), 2014, to appear.

[57] J. J. Dolado, “A Validation of the Component-Based Method for Software Size Estimation,” IEEE Trans. Software Eng., vol. 26, no. 10, pp. 1006–1021, 2000.

[58] A. Corazza, S. D. Martino, F. Ferrucci, C. Gravino, F. Sarro, and E. Mendes, “Using tabu search to configure support vector regression for effort estimation,” Empirical Software Engineering, vol. 18, no. 3, pp. 506–546, 2013.

[59] S. Gueorguiev, M. Harman, and G. Antoniol, “Software Project Planning for Robustness and Completion Time in the Presence of Uncertainty Using Multi Objective Search Based Software Engineering,” in Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, ser. GECCO ’09. ACM, 2009, pp. 1673–1680.

[60] J. D. Kiper, M. S. Feather, and J. Richardson, “Optimizing the V&V Process for Critical Systems,” in Proceedings of the 9thAnnual Conference on Genetic and Evolutionary Computation, ser. GECCO ’07. New York, NY, USA: ACM, 2007, pp. 1139–1139.

[61] F. Ferrucci, M. Harman, J. Ren, and F. Sarro, “Not going to take this anymore: multi-objective overtime planning for software engineering projects,” in ICSE, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds. IEEE/ ACM, 2013, pp. 462–471.

[62] V. Cortellessa, F. Marinelli, and P. Potena, “An optimization framework for ”build-or-buy” decisions in software architecture,” Computers & OR, vol. 35, no. 10, pp. 3090–3106, 2008.

[63] A. Sharma and D. S. Kushwaha, “Applying requirement based complexity for the estimation of software development and testing effort,” SIGSOFT Softw. Eng. Notes, vol. 37, no. 1, pp. 1–11, Jan. 2012.

[64] D. G. e Silva, M. Jino, and B. T. de Abreu, “Machine Learning Methods and Asymmetric Cost Function to Estimate Execution Effort of Software Testing,” in ICST, 2010, pp. 275–284.

[65] X. Zhu, B. Zhou, L. Hou, J. Chen, and L. Chen, “An experience based approach for test execution effort estimation,” in ICYCS, 2008, pp. 1193–1198.

[66] A. Sharma and D. Kushwaha, “A metric suite for early estimation of software testing effort using requirement engineering document and its validation,” in Computer and Communication Technology (ICCCT),2011 2nd International Conference on, 2011, pp. 373–378.

[67] M. E. Helander, M. Zhao, and N. Ohlsson, “Planning models for software reliability and cost,” IEEE Trans. Software Eng., vol. 24, no. 6, pp. 420–434, 1998.

[68] P. A. Scarf, R. Dwight, and A. Al-Musrati, “On reliability criteria and the implied cost of failure for a maintained component,” Rel. Eng. & Sys. Safety, vol. 89, no. 2, pp. 199–207, 2005.

[69] C.-T. Lin, “Enhancing the accuracy of software reliability prediction through quantifying the effect of test phase transitions,” Applied Mathematics and Computation, vol. 219, no. 5, pp. 2478–2492, 2012.

[70] C.-Y. Huang, S.-Y. Kuo, and M. R. Lyu, “An Assessment of Testing-Effort Dependent Software Reliability Growth Models,” IEEE Transactions on Reliability, vol. 56, no. 2, pp. 198–211, 2007.

[71] C.-Y. Huang and J.-H. Lo, “Optimal resource allocation for cost and reliability of modular software systems in the testing phase,” Journal of Systems and Software, vol. 79, no. 5, pp. 653 – 664, 2006.

[72] C.-Y. Huang and M. R. Lyu, “Optimal testing resource allocation, and sensitivity analysis in software development,” IEEE Transactions on Reliability, vol. 54, no. 4, pp. 592–603, 2005.

[73] S. Rafi and S. Akthar, “Incorporating fault dependent correction delay in srgm with testing effort and release policy analysis,” in Software Engineering (CONSEG), 2012 CSI Sixth International Conference on, 2012, pp. 1–6.

[74] O. Berman and M. Cutler, “Resource allocation during tests for optimally reliable software,” Computers & OR, vol. 31, no. 11, pp. 1847– 1865, 2004.

[75] V. Cortellessa, F. Marinelli, R. Mirandola, and P. Potena, “Quantifying the influence of failure repair/mitigation costs on service-based systems,” in 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013, 2013, to appear.

[76] S. Wagner, “Towards Software Quality Economics for Defect-Detection Techniques,” in Software Engineering Workshop, 2005. 29th Annual IEEE/NASA, April 2005, pp. 265–274.

[77] N. Mead, J. Allen, W. Conklin, A. Drommi, J. Harrison, J. Ingalsbe, J. Rainey, and D. Shoemaker, “Making the Business Case for Software Assurance (CMU/SEI-2009-SR-001),” Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2009., [Online]. Available: http://resources.sei.cmu.edu/library/assetview .cfm?AssetID=8831, Tech. Rep.

[78] T. Moser, S. Biffl, and D. Winkler, “Process-driven Feature Modeling for Variability Management of Project Environment Configurations,” in Proceedings of the 11th International Conference on Product Focused Software, ser. PROFES ’10. ACM, 2010, pp. 47–50.

[79] L. Fernndez-Sanz, M. Villalba, J. Hilera, and R. Lacuesta, “Factors with negative influence on software testing practice in spain: A survey,” in European Software Process Improvement Conference (EuroSPI), 2009.

[80] A. Wood, “Software Reliability Growth Models: Assumptions vs. Reality,” in Proceedings of the Eighth International Symposium on Software Reliability Engineering, ser. ISSRE ’97. IEEE Computer Society, 1997, pp. 136–.

[81] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, “Predicting Faults from Cached History,” in Proceedings of the 29th International Conference on Software Engineering, ser. ICSE ’07. IEEE Computer Society, 2007, pp. 489–498.

[82] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” in Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on, May 2005, pp. 284–292.

[83] A. Hassan and R. Holt, “The top ten list: dynamic fault prediction,” in Software Maintenance, 2005. ICSM’05. Proceedings of the 21st IEEE International Conference on, Sept 2005, pp. 263–272.

[84] T. Ostrand, E. Weyuker, and R. Bell, “Predicting the location and number of faults in large software systems,” Software Engineering, IEEE Transactions on, vol. 31, no. 4, pp. 340–355, April 2005.

[85] D. Radjenovic, M. Hericko, R. Torkar, and A. Zivkovic, “Software fault prediction metrics: A systematic literature review,” Information & Software Technology, vol. 55, no. 8, pp. 1397–1418, 2013.

[86] T. McCabe, “A Complexity Measure,” Software Engineering, IEEE Transactions on, vol. SE-2, no. 4, pp. 308–320, Dec 1976.

[87] S. Chidamber and C. Kemerer, “A metrics suite for object oriented design,” Software Engineering, IEEE Transactions on, vol. 20, no. 6, pp. 476–493, Jun 1994.

[88] T. M. Khoshgoftaar, K.Gao, and A. Napolitano, “A Comparative Study of Different Strategies for Predicting Software Quality,” in SEKE. Knowledge Systems Institute Graduate School, 2011, pp. 65–70.

[89] K. Gao, T. M. Khoshgoftaar, H. Wang, and N. Seliya, “Choosing software metrics for defect prediction: an investigation on feature selection techniques,” Software: Practice and Experience, vol. 41, no. 5, pp. 579–606, 2011.

[90] V. R. Basili, G Caldiera, G., and H. D. Rombach, “The Goal Question Metric Paradigm”, in Encyclopedia of Software Engineering, John Wiley & Sons, Inc., pp. 528-532, 1994.

[91] R. Solingen and E. Berghout, “The Goal/Question/Metric Method: a Practical Guide for Quality Improvement of Software Development”, McGraw-Hill, Maidenhead. 1999.

[92] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, “A Systematic Literature Review on Fault Prediction Performance in Software Engineering,” IEEE Trans. Software Eng., vol. 38, no. 6, pp. 1276–1304, 2012.

[93] C. Catal and B. Diri, “A systematic review of software fault prediction studies,” Expert Syst. Appl., vol. 36, no. 4, pp. 7346–7354, 2009.

[94] V. R. Basili, L. C. Briand, and W. L. Melo, “A Validation of Object-Oriented Design Metrics as Quality Indicators,” IEEE Trans. Software Eng., vol. 22, no. 10, pp. 751–761, 1996.

[95] D. Hosmer and S. Lemeshow, Applied Logistic Regression. Wiley-Interscience, 1989.

[96] T. Gyimóthy, R. Ferenc, and I. Siket, “Empirical validation of object oriented metrics on open source software for fault prediction,” IEEE Trans. Software Eng., vol. 31, no. 10, pp. 897–910, 2005.

[97] M. Lorenz and J. Kidd. Object Oriented Metrics. Englewood, NJ: Prentice Hall, 1994.

[98] P. Tomas, M. Escalona, and M. Mejias, “Open source tools for measuring the Internal Quality of Java software products. A survey,” Computer Standards & Interfaces, vol. 36, no. 1, pp. 244 – 255, 2013.

[99] V. R. Basili, L. Briand, and W. Melo, “A Validation of OO Design Metrics as Quality Indicators,” Technical Report CS-TR-3443, 1995.

[100] B. Kitchenham, S. L. Pleeger, and N. Fenton. Towards a Framework for Software Measurement Validation. IEEE Transactions on Software Engineering, Vol. 21, No. 12, pp 929-944,December 1995

[101] N. E. Fenton, “Software measurement: a necessary scientific basis,” IEEE Transactions on Software Engineering, vol. 20, no. 3, pp. 199–206, 1994.

[102] N. E. Fenton, S. Lawrence Pfleeger. “Software Metrics. A rigorous & Practical Approach. 2nd Edition,” ITP, International Thomson Computer Press, 1997

[103] N. F. Schneidewind, “Methodology for validating software metrics,” IEEE Transactions on Software Engineering, vol. 18(5), pp. 410–422, 1992.

[104] P. T. Devanbu, “GENOA: A Customizable Language- and Front-end Independent Code Analyzer,” in Proceedings of the 14th International Conference on Software Engineering, ser. ICSE ’92. ACM, 1992, pp. 307–317.

[105] R. Ferenc and A. Beszedes, “Data exchange with the columbus schema for c++,” in Software Maintenance and Reengineering, 2002. Proceedings. Sixth European Conference on, 2002, pp. 59–66.

[106] “Homepage of FrontEndART Software Ltd.” [Online]. http://www.frontendart.com.

[107] M. D’Ambros, M. Lanza, and R. Robbes, “An extensive comparison of bug prediction approaches,” in Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on, May 2010, pp. 31–41.

[108] F. Brito e Abreu, W. Melo, "Evaluating the impact of object-oriented design on software quality," Software Metrics Symposium, 1996., Proceedings of the 3rd International , vol., no., pp.90,99, 25-26 Mar 1996, doi: 10.1109/METRIC.1996.492446

[109] S. T. S. Demeyer and S. Ducasse, “FAMIX 2.1 The FAMOOS Information Exchange Model,” University of Bern, 2001, Tech. Rep.

[110] R. Moser, W. Pedrycz, and G. Succi, “A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction,” in Proceedings of the 30th International Conference on Software Engineering, ser. ICSE ’08. ACM, 2008, pp. 181–190.

[111] T. Zimmermann, R. Premraj, and A. Zeller, “Predicting Defects for Eclipse,” in Proceedings of the Third International Workshop on Predictor Models in Software Engineering, ser. PROMISE ’07. IEEE Computer Society, 2007, pp. 9–.

[112] T. Graves, A. Karr, J. Marron, and H. Siy, “Predicting fault incidence using software change history,” Software Engineering, IEEE Transactions on, vol. 26, no. 7, pp. 653–661, Jul 2000.

[113] H. Gall, M. Jazayeri, and J. Krajewski, “CVS release history data for detecting logical couplings,” in Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of, Sept 2003, pp. 13–23.

[114] J. Ratzinger, M. Pinzger, and H. Gall, “EQ-Mine: Predicting Short-Term Defects for Software Evolution,” in FASE, ser. Lecture Notes in Computer Science, M. B. Dwyer and A. Lopes, Eds., vol. 4422. Springer, 2007, pp. 12–26.

[115] A. Hassan, “Predicting faults using the complexity of code change” in Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on, May 2009, pp. 78–88.

[116] K. Gokhale, M. Lyu and K. Trivedi, "Incorporating fault debugging activities into software reliability models: A simulation approach," *IEEE Transaction on Reliability*, vol. 2, 2006.

[117] N. Nagappan, T. Ball and A. Zeller, "Mining metrics to predict component failures," *Proceedings of teh 28th International conference on Software engineering*, 2013.