Quantitative Framework for Managing Software Life Cycle

Anca-Juliana Stoica[1], Prabhu Babu[2,*] and Peter Stoica[2]

[1]*Department of Computer and System Sciences, School of Information and Communication Technology, Royal Institute of Technology, Forum 100, S-164 40 Kista, Sweden.* [2]*Dept. of Information Technology, Uppsala University, Uppsala, SE 75105, Sweden*

Abstract: The paper presents a rigorous and practical way of managing the quantitative yields of new (improved) software engineering methods in the software life cycle. A meaningful set of metrics and models is used to measure the value of applying tailored methods. The value-based framework is proposed as a system of integrated models (such as cost models, productivity models, quality-related models, benefit models, and value-related models) that combine project data and expert opinion. The framework is proactive, as it allows to estimate the value-based metrics of a software engineering method in order to monitor it, improve software quality and convince developers and managers that the method and related investment are worthwhile. To demonstrate the applicability of the proposed framework, the results of a case study are presented and used as an initial validation of the framework.

Keywords: Software engineeiring models, quantitative models, model systems, software engineering methods, software measurement and metrics, decision framework, value-based framework.

1. INTRODUCTION

Exploring quantitative yields in the software life cycle when introducing new (improved) software engineering methods is a complex problem that needs to be addressed in a systemic manner [1]. Relevant metrics and models have to be selected, and their interrelationships have to be expressed in a formal manner in order to detect possible inconsistencies or incompletenesses. We defined a model system associated with a particular software product life cycle as a collection of models and the interrelations between them. A model system will therefore contain all the relationships and constraints between models and model elements contained in different models. We used this concept in [1] to analyze multiple facets of software development, and to derive a decision framework for reasoning and value-based decision-making in the software process [2].

In this paper we use the model system concept to address the problem of quantifying the yields of new (improved) software engineering methods and suggest a value-based framework (VF). VF entails the application of interrelated metrics and models from [3-7] for estimating the relevant costs, the benefits translated into software quality management and improvement, as well as the resulting value-related metrics evaluated during the software system life cycle. The framework is applicable at a high level of abstraction and offers useful results from a practical point of view, as our case study indicates.

The paper shows how metrics and models are connected using a rigorous, consistent mathematical approach. Models are easy to adapt to different situations and computational programs re-run to get the numerical and graphical results. Also, the models are applied in a logical sequence by combining project data and expert estimates within a value framework. The metrics and models have been used under actual project conditions: i) with data realistically collected in an Experience Database, and ii) using the output provided by static code analysers and formal inspections. The results of a case study are presented where quantitative yields of the formal inspection process are combined with the application of the latest software quality standards (SQuaRE series) in a hybrid software engineering method that combines agility and discipline based on software quality management. The results are obtained in a software division related to mobile communications which is part of an international company operating in Nordic and Baltic countries. Lessons learned are drawn and presented in the conclusion section.

The paper is structured as follows. Section 2 is dedicated to software engineering models, model systems, and frameworks. Section 3 is a survey on existing software engineering methods and some of the trends for the 21st century. Section 4 connects the two previously mentioned sections using relevant high level metrics and measurements for the software engineering field which are suited for our purpose of quantifying the yields in the software life cycle. Section 5 presents quantitative models for estimating the yields of applying software engineering methods. Section 6 presents the results of a case study in which the models and metrics are used for estimating the value of introducing new (improved) software engineering methods. Finally Section 7 draws the lessons learned and concludes the paper.

2. SOFTWARE ENGINEERING MODELS, MODEL SYSTEMS AND FRAMEWORKS

In the last two decades, it has been largely accepted that software engineering is about producing models.

*Address correspondence to this author at the Dept. of Information Technology, Uppsala University, Uppsala, SE 75105, Sweden;
Tel: (46) 18-471-3394; Fax: (46) 18-511925; Email: prabhu.babu@it.uu.se

1874-107X/11

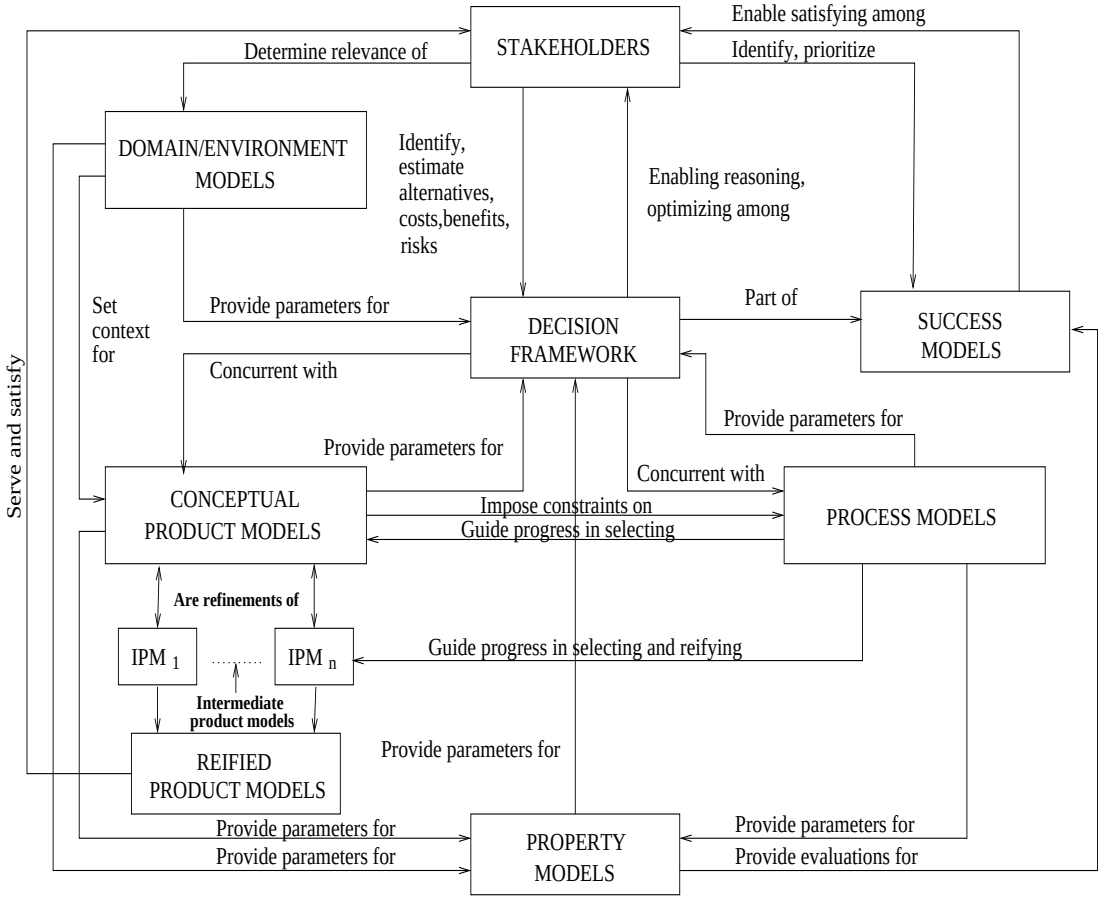

Fig. (1). Model system.

Researchers and practitioners from academia, industry, or standardization organizations have developed and applied software engineering models that are covering a broader scope and are supposed to work interconnected in various modes. Examples are: [8-12].

Identifying the relevant models (and the relevant aspects to be considered) is an important task. Regarding the technical software aspects, the OMG's Model Driven Architecture view [12], for example, is based on the Platform Independent Model and the Platform Specific Model. So the platform is the relevant aspect which corresponds more or less to a type of middleware (like an Object Request Broker, a Hub, or an Application Server). Another example of approach we can refer to is the component-based view [13] which considers architecture as the central aspect.

The views cope with the complexity of software systems and represent abstractions of relevant information for the models. Many other views are available to represent the multiple challenging facets of modern software systems. Frameworks are needed to integrate and analyse views (models). Redundant information is used to verify consistency and completeness between views.

Software engineering models associated with a particular software product life cycle have to satisfy the definition of a system, namely a collection of interrelating parts which, when taken together, form a whole, having properties which cannot be found in the constituent elements. We defined a model system [1] associated with a particular software product life cycle as a collection of models and the interrelations between them (Fig. **1**). A model system will therefore contain all the relationships and constraints between models and model elements contained in different models. We defined a model in a similar manner as in [8], namely a pattern of something to be made, a representation or an analogy used to visualize and reason about the system to be developed and maintained and its likely effects. The model system represented in Fig. (**1**), shows that during software system life-cycle, there are many *stakeholders* like : system developers, acquirers, users, maintainers that are involved in defining, developing, and eventually running the system. Each stakeholder views the system from its own perspective depending on his role and degree of involvement in the system development and operation. In Fig. (**1**), *software engineering models* of increasing level of detail and faithfulness allow compliance of the stakeholders' demands throughout the software system life cycle. These models can be classified into the following categories : product, process, property, and success models. *Product models* include: conceptual product models, intermediate product models (IPMs), and reified product models as ways of specifying operational concepts, requirements, architectures, designs, and code, along with their interrelationships. Reified product models serve and satisfy the stakeholders. *Domain/environment models* from Fig. (**1**), set context for conceptual product models and provide parameters for property models. Stakeholders determine relevance of

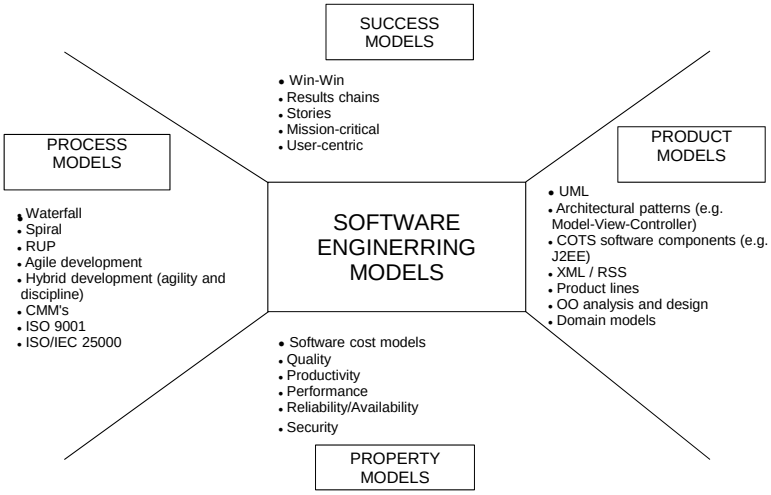

Fig. (2). Examples of models included in a model system.

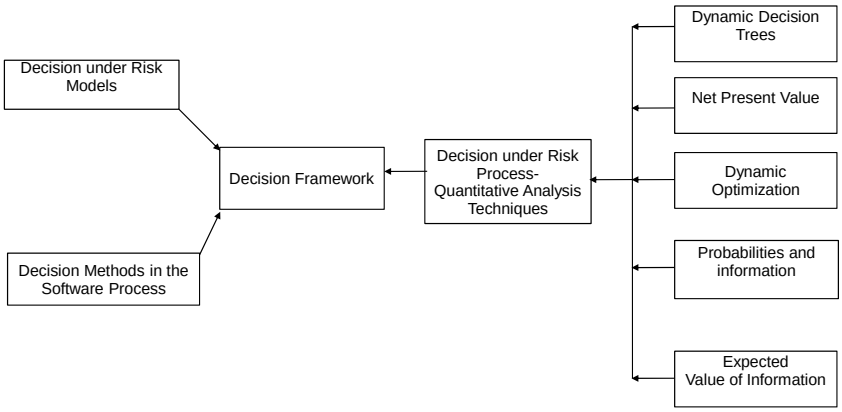

Fig. (3). Components of decision framework.

domain/environment models. *Process models* are used by software developers to create software. Examples are : waterfall model, evolutionary developement model, spiral development, rapid application development, iterative development, agile development, hybrid development. Process models guide progress in : i) selecting conceptual product models ; ii) selecting and reifying intermediate product models. *Property models* define the desired or acceptable level and permissible trade offs for project factors such as : cost, schedule, performance, reliability/availability, security, portability, evolvability and reusability. Provide evaluations for success models. *Success models* examples are: stakeholder win-win, correctness proofs, business case, results chains, stories, mission critical, IKIWISI (I'll know it when I see it), user-centric. Stakeholders identify and prioritize success models. Success models enable satisfying among stakeholders. *Sofware system life cycle* includes : requirements, design, implementation, testing, maintenance. Other components are : forecasting (risk analysis, estimation, planning), learning (process simulation, decision analysis, problem solving, training). *Decision framework* from Fig. (**1**), focuses on the complex problem of value-based reasoning and optimal dynamic decision making among stakeholders in the software engineering process. Process, domain/environment, product, and property models provide parameters for the decision framework. Decision framework is part of success models and it is concurrent with process and conceptual product models. Stakeholders identify and estimate alternatives, costs, benefits and risks used by the decision framework.

In Fig. (**2**) examples of software engineering models included in a model system are given. The terms "model" and "meta-model" are interchangeably used in this paper. It is only the level of abstraction that makes the difference. Model systems define also a notational and semantic integration. Semantic integration means: i) what information; ii) how it can be exchanged; and iii) how to detect inconsistencies between interconnected models.

We used the software engineering models in a broader context, to incorporate aspects of inter-disciplinary nature from statistical decision theory, information theory, utility theory, economics, and optimization theory to derive a theoretical value-based reasoning and decision framework, [2], that can be used to dynamically optimize decision making in the software process when considering the risks associated with the states of the world. Decision framework is also an example of using interconnected models, such as: dynamic decision trees, value of information, net present value, and dynamic optimization with a net value criterion defined as difference between the expected benefits and the costs, evaluated over a defined time horizon. Fig. (**3**) is a graphical representation of the decision framework [2].

Table 1. Complex Problems and Associated Solution Frameworks

Complex Problem	Solution Framework
Value-based reasoning and optimal decision making in software engineering process	Decision framework
Managing the quantitative yields of software engineering methods in software life cycle	Value-based framework

Table 2. Solution Frameworks and Associated Aspects/Models

Solution Framework	Associated Aspects/Models
Decision framework	DDT, VOI, NPV, DO, NV
Value framework	CM, WM, QM, BM, VM

In this paper, for managing and quantifying the yields of software engineering methods, we show that identifying core technical and economic aspects are relevant for model selection, like quality, productivity, costs, benefits, and value. Having a formal description of these aspects helps to define their scope, to understand their essence as well as their interactions. This is also another example of our model system concept. We consider these models as parts of a *value framework* that we define as a collection of interconnected models such as: cost models, productivity models, quality-related models, benefit models, value-related models, evaluated over the software life cycle when using software engineering methods.

We use a model for each recurrent technical or economic aspect of software, like quality, productivity, costs, benefits, and value. These concepts and models are also equivalent with the non-functional or emergent system properties that depend on the software engineering method used for software system development. We also call them "property models" in the model system concept.

The two complex problems that are solved applying the model system concept and their associated solution frameworks, are shown in Table **1**.

The two solution frameworks have a number of interrelated aspects and models, that are shown in Table **2**.

Problems from Table **1** may arise at a certain stage in the software engineering process, namely at: i) a high level of abstraction or software architecture level, or at: ii) a low level of abstraction or detailed software implementation level. Here we propose the use of high-level of abstraction models to solve our problem from a practical point of view.

As an example, we use the model system concept instantiation from Fig. (**1**) which includes decision framework and value framework applied to web application projects :

- *Stakeholders*: software development team, quality assurance team, project manager, clients
- *Decision framework*: value-based analysis and optimization of time-related major software engineering or re-engineering investment decisions
- *Success models*: Win-Win [14], agility, quality assurance (like achieving high usability, maintainability, availability, efficiency)
- *Domain/environment models*: set the context for conceptual product models. Example: Internet communities of WAP/GPRS mobile phone users that need new (high quality) multimedia services
- *Conceptual/product models*: architectural patterns e.g. Model-View-Controller, commercial off-the-shelf components (e.g. J2EE components), schedule-constraint systems
- *Process models*: feature-driven, iterative, balancing agility and discipline, formal software inspections
- *Property models*: quality, productivity, costs, benefits, and value models included in a value-based framework.

In Sections 4, 5, and 6 of this paper, we present relevant quantitative metrics and models for the value-based framework that satisfy the semantic integration principles and their application for two projects: a reference project and a real case project.

3. SPECTRUM OF SOFTWARE ENGINEERING METHODS

Software engineering methods are structured approaches to software development that include:

- design advice
- process guidance
- modeling approaches, usually visual and conceptual,

whose aim is to facilitate the production of high-quality software in a cost-effective way. However, the software engineering methods of today do not include quantitative modeling approaches for estimating the IT yields in the software life cycle. In this paper we use modeling concepts and high-level models to quantitatively explore the yields of applying software engineering methods.

A possible way of analyzing the current software engineering methods, presented in [15], is by considering their spectrum represented on a horizontal axis that is

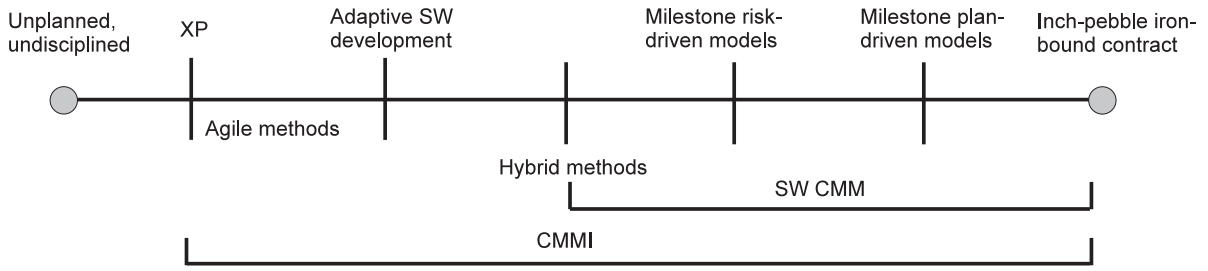

Fig. (4). The spectrum of software engineering methods, adapted from [15].

associated (from left to right) with increased planning effort (Fig. **4**).

Examples of software engineering methods from the spectrum follow :

Agile methods - adaptive, rather than predictive; people-oriented rather than process-oriented; code-oriented methods; have different approaches, such as: Agile Modeling [16]; Crystal Methods [17]; Dynamic Systems Development Method [18]; Feature-Driven Development Method [19]; Scrum [20]; Extreme programming [21].

Adaptive Software Development [22]: center-left; based on the principle of continuous adaptation of the process to the work at hand.

Milestone risk-driven methods examples are: Unified Software Development Process, Rational Unified Process [23]; Model-Based Architecting and Software Engineering, based on the Spiral Model [8]. These methods incorporate software engineering practices; the time dimension is associated with the Inception, Elaboration, Construction, and Transition phases, their associated major milestones (or anchor points), and their respective iterations; provide guidelines for applying software engineering methods that are characterized by providing support for both discipline and flexibility using software risk management.

Milestone plan-driven approaches examples are: Personal Software Process / Team Software Process [24, 25], Cleanroom [26], Capability Maturity Model [27], ISO/IEC software quality management standards [28-30].

Personal Software Process / Team Software Process are structured frameworks of: forms, guidelines, and procedures for developing software; Personal Software Process is directed towards the use of self-measurement to improve individual programming skills; Team Software Process builds on Personal Software Process and supports the development of industry-strength software done by team planning and control.

Cleanroom method uses statistical process control and mathematics-based verification to develop software with certified reliability.

Software Capability Maturity Model (SW CMM) is a software engineering method approach, based on highly disciplined software processes in aerospace and commercial industries; process improvement framework grew out of the need for Air Force to select qualified software developers; collects best practices into Key Practice Areas that are organized into 5 levels of increased maturity; focuses on project management, rather than product development; no rapid application development techniques; no risk management as key process area.

Capability Maturity Model Integrated (CMMI) - covers a broader part of the method spectrum and includes: integration of software and system CMMs; suite of models and appraisal methods that address a variety of disciplines using common architecture, vocabulary, and core of process areas; process areas for risk management, integrated teaming, and an organizational environment for integration, to include agile methods.

International Organization for Standardization (ISO) - the ISO 9001 Quality Management Systems - Model for Quality Assurance in Design/Development, Production, Installation, and Service [30], followed by ISO/IEC software quality management and process standards like : ISO/IEC 90003 [28], and ISO/IEC 25000, the SQuaRE (Software Quality Requirements and Evaluation) series [29] .

Besides the agile, milestone risk-driven, and plan-driven methods we can also mention:

Hybrid methods [31, 32] that combine agility and discipline using software risk management and a unified process process framework to tailor risk-based processes into an overall development strategy.

The method application depends on the type of product to be developed and maintained.

- For large systems, a strictly managed process is needed.
- For smaller systems, more informality is possible, without neglecting quality assurance aspects, which we consider a future need (besides rapid value) to gain competitive advantage for software organizations.

Our experience also shows that there is no uniformly applicable method that should be standardized within an organization. High costs and low value may be incurred if an inappropriate method is forced on a development team.

Future trends for the 21st century processes and methods that we can mention here are:

a) In the area of developing large- to very large scale software-intensive systems and systems-of-systems, the Emerging Scalable Spiral process model described in [33] will cope with increasing integration of software and system engineering; emphasis on users and end value, dependability, rapid change, global connectivity, interoperability; needs for COTS, reuse, legacy system and software integration; and computational plenty.

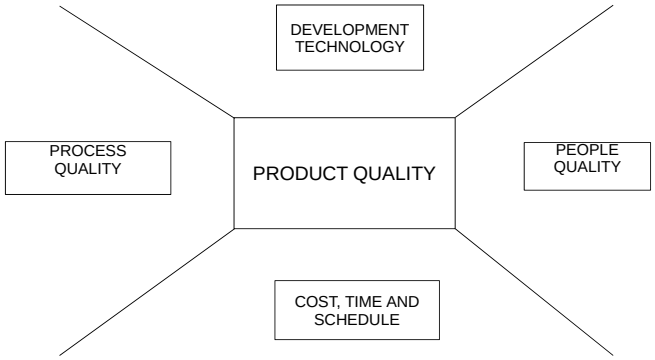

Fig. (5). Product quality factors.

b) In the area of developing small- to medium scale software systems, future methods will possibly be of a hybrid nature, in order to scientifically combine agility and discipline using in addition to software risk management [32], some other principles, like software quality management.

In order to get competitive advantage on the market, rapid value will not be sufficient, and high assurance compliant with the emerging software quality standards will be a good solution [34].

Our paper contains results on exploring quantitative yields when using an improved software engineering method. The method used is hybrid software system development and evolution with emphasis on quality assurance. Hybrid approaches are feasible and necessary for projects that have a mix of characteristics, like Project 2 from Section 6. Fig. (**5**) shows the product quality factors considered here.

4. METRICS AND MEASUREMENTS IN THE SOFTWARE ENGINEERING FIELD

In this section, we introduce metrics and measurements that are relevant in our context. The metrics have to combine the software engineering know-how of measurement experts with the domain know-how of developers. Our metrics will refer to software system process, product, and property metrics.

4.1. General (Basic) Metrics

In order to explore the quantitative yields in the software life cycle, we identified a number of basic metrics as follows :

Size (S) related to a software product that can be estimated as: source lines of code [SLOC], function points [*FP*], object points [OP], or use-case points [UCP].

Effort (*E*) - for software development or maintenance estimates how many *person. months*, [*PM*] or person. hours, [*PH*] are needed for the development or maintenance stages of the software life cycle:

$$E = f(S) \tag{1}$$

Cost (*C*) - is also associated to development or maintenance stages of the software life cycle:

$$C = E * c \tag{2}$$

where:

- c - unit labor cost, [*m.u./ PH*], m.u.=monetary units.

Schedule (T) - measures the calendar time a process requires, in [months], or [M]:

$$T = g(E) \tag{3}$$

Productivity (W) - measures software process output per unit of effort [$output / PM$], under a very important assumption, namely that the project is well-managed and does not make a wasteful use of resources. This is an important responsibility for the project manager and all the project members :

$$W = \frac{S}{E} \tag{4}$$

Quality (Q) - can be measured with respect to the number of defects a process yields, relative to the process output, and is given by:

$$Q = (1 - DD) * 100, [\%], \tag{5}$$

where:

- *DD*- defect density [defects/SLOC].

There are many definitions of quality. Sometimes, quality is associated with a large number of attributes, like in the software quality standards models, such as the ISO/IEC software quality SQuaRE series.

DD from (5) is not only used for measuring code quality, but it is also used for managing software projects, as it was done in our case study from Section 6.

Reliability, (*R*), is the probability that the system will not fail per given unit of time.

Average Staffing, (*A*), in full-time software personnel, [*FSP*], for development or for maintenance is given by:

$$A = \frac{E}{T} \tag{6}$$

Documentation, (DOC) [pages] is estimated as a function of the software size:

$$DOC = h(S) \tag{7}$$

Remark: Functions (*f*), (*g*), (*h*) from (1), (3), (7) are determined by applying: i) the algorithmic cost modeling approaches described in [3, 5]; ii) expert judgement, and iii) estimation by analogy.

Performance, (*P*) or effectiveness of a software system measures the output (e.g.: transactions, for a transaction processing system) per unit of time: P[tr/sec].

Efficiency - measures system performance or effectiveness relative to resources consumed.

Customer Satisfaction - measures how well the clients are served, and can be expressed using ratings that are associated with a numerical scale.

Innovation - for modern software products, like Web-based applications, measures the range and creativity of

products and services that are offered to the clients. Can be measured in the same manner as the customer satisfaction.

4.2. Specific Quality-Related Metrics

We focus on the following relevant metrics :

Defect Density, (DD), is measured by the number of defects *ND* relative to the software size *S*, [defects/SLOC]:

$$DD = \frac{ND}{S} \tag{8}$$

The defects remaining in the software after the project was completed can be found in: requirements, designs, code, or test cases. Therefore, *DD* it is not only a measure of code quality, but also a way of properly managing software projects.

Defect Removal Efficiency, (DRE) in [%] is given by:

$$DRE = \frac{DR}{DE + DI} * 100, [\%], \tag{9}$$

where:

- *DR*- number of defects removed
- *DE*- number of defects inherited (existent)
- *DI*- number of defects injected

DRE is connected to a specific software appraisal activity, such as individual reviews, testing, automatic static and dynamic code analysers, etc. In Section 6 we present the results obtained by using models for evaluating the quantitative yields in software life cycle, generated by applying an improved software engineering method, and using the above mentioned software appraisal activities. *DRE* can be applied on a phase-by-phase basis, or activity-by-activity basis in order to evaluate the effectiveness of a phase or individual activity in the software process.

Defect Removal Model - (DRM), $[defects]$, is given by:

$$DRM = DE + DI - DR \tag{10}$$

An important observation here which connects metrics from subsections 4.A and 4.B is that effort, cost, and schedule (*E*, *C*, *T*) basic metrics are linked to the software quality metrics, in the sense that if software quality targets are not achieved, all the three basic metrics will be exceeded significantly.

Measurements in software engineering and the associated quantitative models and analyses are an integrated discipline, rather than stand-alone processes. Some relevant examples are: software quality, software reliability or availability, software structure and design metrics, software system cost estimation, or software security metrics.

Despite its importance, most software organizations do not perform any software engineering measurement at all, and do not collect their past project data in experience databases in order to use quantitative models and analyses and gain competitive advantage on the market. Only world-class software organizations or organizations that aim at achieving peak operational efficiency and competitivity are using these techniques. Project from Section 6 is an example of software development in such an organization.

5. QUANTITATIVE MODELS FOR ESTIMATING COSTS, BENEFITS, AND VALUE OF SOFTWARE ENGINEERING METHODS

In this section we present a value-based framework consisting of related quantitative models for estimating costs, benefits, and business value of a subset of software engineering methods that share common characteristics such as: i) emerging and industry standards for improving product quality, productivity, and performance, as well as software management performance; ii) defined, repeatable, measurable, highly beneficial, and in common use; iii) requiring specific training to apply and relatively expensive to use.

The framework is instantiated for the formal software inspection method (SIM) as being one of the most used software engineering method. SIM consists of special type of meetings held to objectively identify the maximum number of defects in software work products and to improve software quality. Basic principles that govern SIM are: technical peers identify defects; defects must be corrected without suggesting solutions or interference from the originator of work product; technical experts cannot suggest design alternatives or subjective improvements to the product.

Formally, SIM consists of six stages:

1) Planning (schedule, announce, coordinate)

2) Overview (communicate, educate, learn)

3) Preparation (study, analyze, examine)

4) Meeting (facilitate, identify, record)

5) Rework (search, repair, finalize)

6) Follow-up (verify, measure, report).

The net result of applying these formal stages is the transition from an initial draft, pre-baselined, and high-defects product, to a final, baselined, and low-defects product. The six stages are designed to: add value, structure, repeatability, measurability; optimize the number of defects identified, thus optimizing software quality.

SIM can be applied: at the end of software life cycle (SLC) phases; after work products have been completed within individual phases; at critically important decision points within SLC; as a scheduled event in software project plans, not as an ad-hoc activity subject to preemption.

5.1. Quantitative Models for Estimating Costs

5.1.1. Cost Components

$$C_{SIM} = C_D + C_M + C_T + C_I + C_{TR} \tag{11}$$

where:

- C_{SIM} - complete cost when applying SIM
- C_D - software development cost
- C_M - software maintenance cost

- C_T - software testing cost
- C_I - cost of implementing SIM
- C_{TR} - cost of training

5.1.2. Total life Cycle Cost

$$TLC_{SIM} = (S * K - T_I * 99 - T_{T,a} * 9) * c \quad (12)$$

where:

- TLC_{SIM} - total software life cycle cost, that includes software development and maintenance costs, and reflects the effects of applying models for estimating: defect density, software quality, defect removal, and defect removal efficiency
- S - estimated software size, [SLOC]
- K - average software development and maintenance effort per SLOC [PH/SLOC], without using formal inspections or testing
- T_I - calculated (planned) time for inspections, [H]
- $T_{T,a}$ - calculated total time for testing, after applying the inspections, [H]
- c - labor cost per hour [m.u./PH], m.u.= monetary units

TLC is based on a model suggested in [7] and applying the economics of SIM, software testing and software maintenance. Conservative assumptions are that a defect may be repaired in: 1 hour using SIM, 10 hours using software testing, and 100 hours using software maintenace. In [7] it is suggested that K=10.51 for modern intermediate- to medium scale software systems, but equation (12) can be calibrated as a function of: development and maintenance effort, defect removal efficiency, and inspection and testing efficiencies.

We estimate first the the components of the total cost, from (11).

Cost of training:

$$C_{TR} = c_{TR} * N \quad (13)$$

where:

- c_{TR} - cost of training per person [m.u./P]
- N - team size for development, persons [P]

Cost of implementing SIM:

$$C_I = (n_M * t_{SIM-run}) * c \quad (14)$$

where:

- n_M - number of meetings to implement SIM
- $t_{SIM-run}$ - time per SIM run (given by the activities included in a SIM run: planning, overviews, preparation, meetings, rework, and follow-up)

Number of meetings to implement SIM:

$$n_M = \frac{S}{ir} \quad (15)$$

where:

- ir - average inspection rate [SLOC/meeting]

Calculated (planned) time for inspections can be estimated using two alternative methods, and we denote the results by $T_I^{(1)}$ and $T_I^{(2)}$:

$$T_I^{(1)} = n_M * t_{SIM-run} \quad (16)$$

$$T_I^{(2)} = \left(\sum_{i=1}^{p} \frac{ps_i}{rr_i * 2} \right) * (N * 4 + 1) \quad (17)$$

where:

- ps_i - work product size per phase (ex.: number of requirements, number of diagrams, number of SLOC, or number of test cases)
- rr_i - review rate per hour and phase (req/H, diagr/H, SLOC/H, or tc/H)

Based on $T_I^{(1)}$ and the time required to find a defect using software inspections, the number of defects that are detected by formal inspections is given by:

$$n_{D,I} = \frac{T_I^{(1)}}{t_{D,I}} \quad (18)$$

where:

- $T_I^{(1)}$ - calculated (planned) time for inspection
- $t_{D,I}$ - average time to find a defect by inspection

If we assume that $n_{D,S}$ is the number of defects existent in the software before inspections (relative to the estimated software size), then the remaining defects after applying the inspections are:

$$n_{D,R} = n_{D,S} - n_{D,I} \quad (19)$$

where:

- $n_{D,R}$ - number of remaining defects after applying SIM
- $n_{D,S}$ - number of defects existent in the software
- $n_{D,I}$ - number of defects detected by SIM.

Now suppose that we apply software testing to deal with the remaining defects, and that the effectiveness of the testing process is estimated by the e_T parameter. Then the number of defects that software testing is detecting is given by:

$$n_{D,T,a} = e_T * (n_{D,S} - n_{D,I}) \quad (20)$$

where:

- $n_{D,T,a}$ - number of defects detected by software testing, after applying the inspections

- e_T - effectiveness of software testing

Assuming that we can estimate an average time to find a defect by testing, $t_{D,T}$, the cost of testing for the remaining defects is given by the formula:

$$C_T = (n_{D,T,a} * t_{D,T}) * c \quad (21)$$

where:

- C_T - cost of testing for the remaining defects
- $t_{D,T}$ - average time to find a defect by testing

If $t_{D,T}$ is the estimated time to find a defect by testing, then the total time for software testing after inspection is:

$$T_{T,a} = n_{D,T,a} * t_{D,T} \quad (22)$$

Now we have all the necessary components to estimate the total life cycle cost by applying (12). The maintenance cost C_M can be obtained using:

$$C_M = TLC_{SIM} - (C_D + C_I + C_T) \quad (23)$$

We have estimated all the cost components C_D, C_M, C_T, C_I, C_{TR} when using the improved software engineering method to help develop a software system:

- C_D is estimated using software cost estimation models and averaging techniques [3-5]
- C_M is estimated using formulas (23), (12), (14), (21)
- C_T is estimated using formula (21)
- C_I is estimated using formula (14)
- C_{TR} is estimated using formula (13).

5.2. Quantitative Models for Estimating Benefits

5.2.1. Gross Benefit

$$GB = TLC_b - TLC_a \quad (24)$$

where:

- *GB*- gross benefit
- TLC_b - total life cycle cost before applying SIM
- TLC_a - total life cycle cost after applying SIM

$$TLC_b = (S * K - T_{T,b} * 9) * c \quad (25)$$

where:

- $T_{T,b}$ - total time for testing without applying SIM

$$TLC_a = (S * K - T_I * 99 - T_{T,a} * 9) * c \quad (26)$$

where:

- $T_{T,a}$ - total time for testing after applying SIM

$T_{T,b}$ is different from $T_{T,a}$ because basically applying SIM reduces the number of defects remained to be detected by testing:

$$T_{T,a} < T_{T,b} \quad (27)$$

If we estimate the following input variables:

- $n_{D,S}$ - number of defects existent in a software system of size *S*
- e_T - effectiveness of software testing
- $t_{D,T}$ - average time to find a defect by testing

then we compute the defects detected by testing before applying SIM as:

$$n_{D,T,b} = e_T * n_{D,S} \quad (28)$$

It follows that $T_{T,b}$ can be calculated as:

$$T_{T,b} = n_{D,T,b} * t_{D,T} \quad (29)$$

We can now apply (25), (26) and calculate total life cycle cost before and after applying the SIM based on:

- T_I - that was previously calculated using (16) or (17)
- $T_{T,a}$ - which is calculated using the same reasoning as $T_{T,b}$, but with the number of defects detected by testing after applying software inspections and given by (22).

So, using (26) we can compute the total life cycle time after applying SIM, and then using both (25) and (26) we get the gross benefit from (24).

5.2.2. Net Benefit

$$NB = GB - AC, \quad (30)$$

where:

- *GB*- gross benefit [m.u.]
- *AC*- additional cost due to using SIM [m.u.]

$$AC = C_I + C_{TR}, \quad (31)$$

where:

- C_I - cost of implementing SIM [m.u.]
- C_{TR} - cost of training [m.u.]

5.3. Quantitative Models for Value Estimation

The following models are useful in this context:

5.3.1. Benefit-to-Cost Ratio

$$BCR = \frac{GB}{AC} \quad (32)$$

where:

- $GB = TLC_b - TLC_a$ is the gross benefit
- *TLC*- total life cycle costs before applying SIM, given by (25)
- *TLC*- total life cycle costs after applying SIM, given by (26)
- *AC* - additional cost, due to using SIM, given by (31).

5.3.2. Return on Investment

$$ROI = \frac{NB}{AC} * 100, [\%] \quad (33)$$

where:

- *NB*- net benefit, given by (30)

5.3.3. Net Present Value

Net present value of gross benefit, or discounted gross benefit is today's value of future gross benefit GB, which is going to be generated during the software life cycle:

$$NPV(GB, r, n) = GB_D = \frac{GB}{(1+r)^n}, \quad (34)$$

where:

- r - discount rate per year
- n - estimated life cycle time, [years]

Based on GB_D, we can also compute the discounted net benefit:

$$NB_D = GB_D - AC, \quad (35)$$

where:

• *AC*- is the additional cost, due to using SIM, given by (31) [m.u.]

The return on investment using the net present value of net benefit is given by:

$$ROI_D = \frac{NB_D}{AC} * 100, [\%] \quad (36)$$

The quantitative model (36) is a more accurate estimate of the ROI, taking into account the time value of money:

$$ROI_D < ROI \quad (37)$$

5.3.4. Breakeven Point

The breakeven point is used for economic forecasting because it shows when the profits will begin to flow, or when profits will be above some level of expenditures (in units of time or units of work products). It helps the decision-making process and allows us to optimize the value of applying a software engineering method. We use the following model for estimating the breakeven point in [m.u.]:

$$BEP = \frac{AC}{1 - W_b / W_a}, \quad (38)$$

where:

- AC - additional cost, due to using SIM, given by (31)
- W_b - software productivity before introducing SIM, [SLOC/PH]
- W_a - software productivity after introducing SIM, [SLOC/PH]

$$W_b = \frac{S}{E_{LC,b}} \quad (39)$$

$$E_{LC,b} = \frac{TLC_b}{c}, \quad (40)$$

where TLC_b is given by (25). Similarly, W_a is the software productivity after introducing SIM:

$$W_a = \frac{S}{E_{LC,a}}, \quad (41)$$

where:

$$E_{LC,a} = \frac{TLC_a}{c}, \quad (42)$$

and TLC_a is given by (26).

Another way of expressing BEP is in units of time, e.g. hours [H], after the project start, when the benefits will become larger than costs:

$$T_{BEP} = \frac{BEP - AC}{N * c}, \quad (43)$$

where:

- *N* is the project team size, persons [P]
- c is the cost per person and hour, [m.u./PH]

5.4. Input Parameters for Using the Quantitative Models

- *S* - software size, [SLOC]
- *N* - average team size for the project, persons [P]
- c_{TR} - cost of training per person, [m.u./P] *
- $t_{SIM-run}$ - time per SIM run, [H/run] *
- *ir* - inspection rate, [SLOC/meeting]
- ps_i - product size per phase (number of requirements, or diagrams, or SLOC, or number of test cases)
- rr_i - review rate per hour and phase [product size/H]
- c - labor cost per hour, [m.u./H]
- $t_{D.I}$ - average time to find a defect using formal inspections, [H/defect]
- $n_{D.S}$ - estimated number of software defects before using SIM, [defects]
- e_T - software testing efficiency, [%]
- $t_{D.T}$ - average time to find a defect by testing, [H/defect]
- E_D - software development effort [PH], obtained from several cost estimation models averaged together using a Delphi method
- *r* - discount rate per year [%]
- *n* - estimated life cycle time [years]
- *T* - schedule [hours]
- n_M - number of meetings [meetings]

*3) "4*4" software project view model*

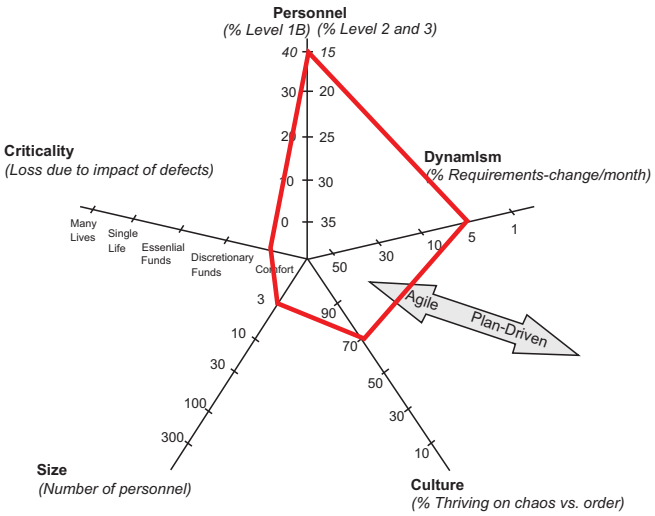

Fig. (6). Case study project dimensions.

- *K*- average software development and maintenance effort per SLOC [PH/SLOC]
- * c_{TR} and $t_{SIM-run}$ are method-specific parameters

6. RESULTS

The models and framework presented in previous sections have been used to manage and evaluate the quantitative yields in the software life cycle when applying new (improved) software engineering methods.

The results are presented for two projects : Project 1 is an illustrative example for our value-based quantitative framework; Project 2 is a case study developed using a hybrid software process balancing agility and discipline with software quality management.

Our framework is instantiated for projects having common characteristics such as: web-based small to intermediate size projects; stakeholder commitment based on Theory W principles; incremental development and delivery; balancing agility and discipline; quality management done through the whole software life cycle; use of a tailored software engineering method.

6.1. Case Study

The company-based case study focus is on managing small agile projects with project deliverables for telecommunication sector users. The company is a leading software development company specialized in software product engineering services with many years of experience with solutions covering the whole product life cycle. The vision is to capture a larger market size and become more attractive to customers by developing better software quality products and managing better software development processes.

1). Project Description

The goal of this project is to publish Web Logs (Blogs), via WAP or GPRS to the customers of software owner (telecommunication company). Publications are delivered to application by RSS (RSS is a family of XML file formats for web syndication used by weblogs) and server creates pages that are suitable for mobile stations to read.

Personnel Details

Three persons: project manager between Cockburns [17] level 1A and 2, developers Cockburns level 1B. Scheduling and planning: iterations-driven development.

Refactoring and re-usage of components was considered when project estimation was done.

Estimating size of source code was done based on previous experience (similar projects). Since a lot of components were the same, the analogy was possible.

Development was feature-driven. Project manager divided tasks (features) and programmers started working on it. Project manager took care of daily problems and most of the customer communication. Programmers and project manager had several meetings in a week (quality related meetings and status meeting). In the beginning of iteration, there was a kick-off meeting, where the requirements were discussed and tasks divided between developers.

Customer was always available. Not on-site, but via messenger, phone or email.

Technical Details

Design was kept as simple as possible. The design was done in the beginning of iteration and it took into consideration only requirements that the iteration had. Testing was scenario based and took place at the end of iteration. Documentation was mainly in company's WIKI (a website that allows the creation and editing of any number of interlinked webpages via a web browser using a simplified markup language). Only important documents were published (Administrator and Configuration Guide, User Guide, Project Quality Evaluation Plan). The source code was committed to SVN repository and it had to compile. The SVN repository always contained the latest working source code, because developers integrated their code one at a time

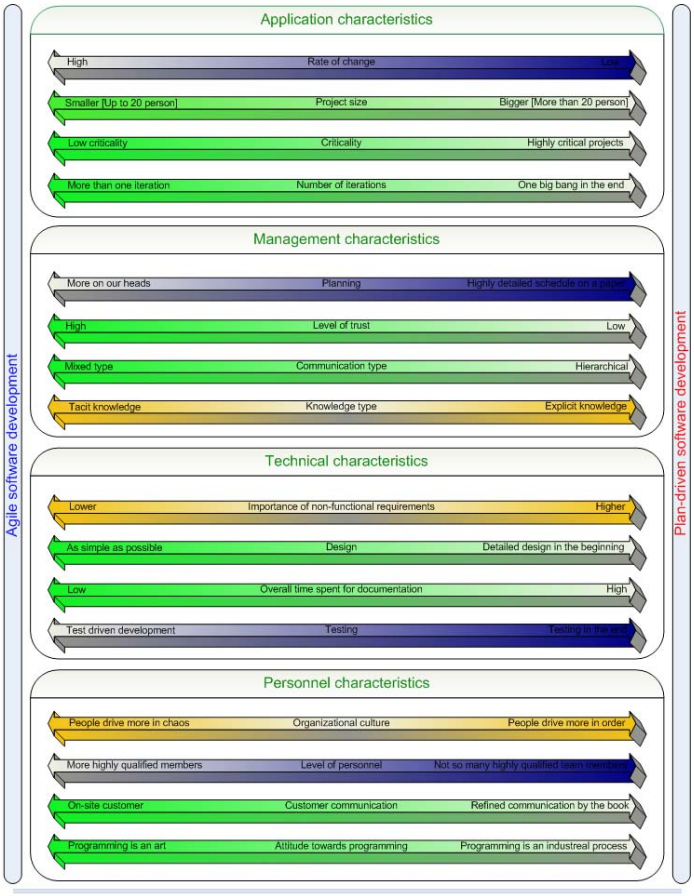

Fig. (7). The ``4*4" software project view model applied to the case study.

Table 3. Estimates Per SIM Run

SIM Stage	Participants	No. of Persons [P]	Time [H/P]
1. Planning	moderator	1	0.5
2. Overview	*all*	*N*	1
3. Preparation	inspectors	*N*-1	1
4. Meeting	*all*	*N*	2
5. Rework	workauthor	1	1
6. Follow-up	moderator	1	0.5

after finishing a feature. The decision was to use SVN, because it is a promising technology and the company vision and goal is to always use the best technology.

2). Polar Chart

The polar chart represents the project dimensions in 5 different areas: size, criticality, personnel, dynamism, and culture and gives a good overview of the project. The main reason for this graphical representation is to see whether the project is characterized as an agile, plan-driven, or hybrid (Fig. **6**).

The five areas correspond to the five critical factors involved in determining the relative suitability of an agile, plan-driven, or hybrid method in a particular project situation and are described in [32].

As seen from the chart in Fig. (**6**), not all results in different axes are close to the centre. It means that this project should use a hybrid software development method to achieve the best results, with application of ISO/IEC SQuaRE quality standards that add plan-driven activities. This is a concise graphical representation of the project dimensions, that has to be accompanied by a list of practices and techniques used in the project as well.

*3). "4*4" Softwear Project View Model*

Our "**4*4**" view model applied to the case study project is shown in Fig. (**7**).

In Fig. (**7**), software project view model has 4 areas: application, management, technical, and personnel

characteristics and each area is divided into 4 sub-characteristics.

A. APPLICATION CHARACTERISTICS

1. Rate of Change

How often the requirements and stakeholders needs change over time?

2. Project Size

Number of persons in the project. The smaller the number the better is to adopt an agile method. If the number of persons is high, scalability is low and vice versa (if the number of persons is low, scalability is high). Different sources use different numbers about the number of participants that is best or the upper limit for agile projects. In case of Crystal Clear agile method [17] the number is up to 8, but [32] describes even an agile project that had 50 persons. This number depends a lot on communication skills of the development team. We suggest 20 people as a reasonable team size to hold a meeting and to have a meaningful communication process.

3. Criticality

How large is the loss due to the impact of software defects?

4. Number of Iterations

If customer has rapid need of software and if there are more than one software releases (product oriented), then it is not one project release, which illustrates most of the plan-driven (process controlled) methodologies.

B. MANAGEMENT CHARACTERISTICS

1. Planning

Does the project have detailed plans in the beginning or it plans iteration by iteration. There are differences between agile and plan-driven software development planning. Plan-driven prefers highly detailed schedules, which cover the whole project. Agile processes prefer planning one iteration at a time. They have general plan for the whole, but they plan only necessary amount.

2. Level of Trust

How much team members trust each other. If project members prefer to get the information on documents (web, specifications, etc.), instead of asking from the others, then it is a sign of not trusting the other one (it also indicates problems with communication and team building).

3. Communication Type

How is team communication specified? If there are separate project teams and each of them has a team leader, then how do members of separate teams communicate? Do they speak to their team leader or directly to each other?

4. Knowledge Type

Tacit knowledge or documented knowledge? Even agilists write documents, but they draw a line between necessary and exhaustive.

C. TECHNICAL CHARACTERISTICS

1. Importance of Non-Functional Requirements

The importance of non-functional requirements shows circumstantially the criticality. It describes how reliable, secure, safe, well performing, usable, efficient, portable and ethical the software system is.

2. Design

Is the design as simple as possible and changeable when necessary or is there an extensive design done at project start ?

3. Overall Time Spent for Documentation

How much time the team spends for documentation and reviews.

4. Testing

Which approach did the project follow: test-driven development (executable test cases) or documented test plans and procedures ?

D. PERSONNEL CHARACTERISTICS

1. Organizational Culture

How does the team perform best: thriving on chaos or on order?

2. Level of Personnel

Has the team more high skill level people or team members have both high and low skill levels?

3. Customer Communication

How and how often did the development team communicate with the customer? Agile methodologies prefer on-site or at least retrievable customer. Plan-driven customers usually are retrievable during requirement specification but not so much after it.

4. Attitude Towards Programming

The development team members view to programming. Are they thinking that programming is an art or developers think that programming is an industrial process?

Table **4** summarizes the “4*4" software project view model instantiated for agile and plan-driven methods.

How to apply the “4*4" software project view model:

In Fig. (**7**), on the left side there are agile characteristics and on the right side plan-driven characteristics. From the project description and for each characteristic in the “4*4” model mark one side that is more appropriate (agile/plan-driven). It is possible to mark both sides as well, if project description requires it. In the end if all choices are made, the “4*4”software project view model will show whether the project will make best use of an agile, plan-driven, or a hybrid method.

The purpose of this model is also to reveal the different facets of the software development project.

After applying the model, it is easy to see that there are agile characteristics and few plan-driven characteristics, thus

Table 4. The ``4*4" Software Project View Model

Software Project Characteristics	Agile Software Development	Plan-Driven Software Development
A Application characteristics		
1 Rate of change	High	Low
2 Project(team) size	Smaller (up to 20 persons)	Larger (more than 20 pers.)
3 Criticality	Low	Highly critical projects
4 Number of iterations	Product-oriented	Given by project plan
B Management characteristics		
1 Planning	Based on practices	Documented project plan
2 Level of trust	High	Low
3 Communication type	Mixed type	Hierarchical
4 Knowledge type	Tacit knowledge	Explicit knowledge
C Technical characteristics		
1 Importance of non-functional requirements	Lower	Higher
2 Design	As simple as possible	Detailed initial design
3 Overall time spent for documentation	Low	High
4 Testing	Test-driven development	Documented throughout the project
D Personnel characteristics		
1 Organizational culture	People thrive more in chaos	People thrive on order (policies and proce-dures)
2 Level of personnel	More highly qualified members	Vary with the project complexity
3 Customer communication	On-site customer	Refined communication by the book (formal relationship with customer)
4 Attitude towards programming	Programming is an art	Programming is an industrial process

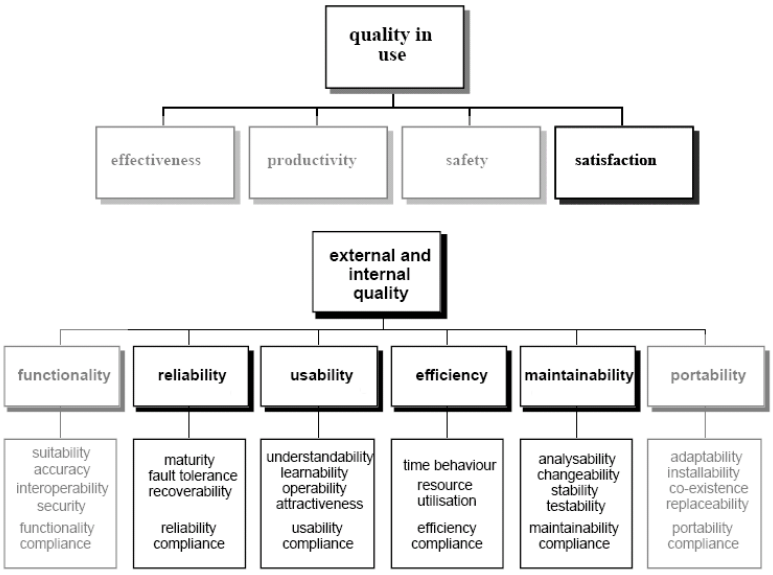

Fig. (8). Project case study quality model.

this project will use a hybrid software development method: agile with influences from plan-driven development.

4). Quality Model

Quality model used is based on the ISO/IEC quality model (see Fig. **8**). Dark colours represent the case study quality model and light grey colours indicate ISO/IEC quality characteristics that were not considered.

5). Techniques and Practices

The real-life case study used a hybrid software engineering method characterized by a number of specific techniques and practices such as:

Iterative Development - Project had several iterations. Iteration length was from 2 weeks to 5 weeks

Small Team - Lead developer, project manager and a developer

B. Quantitative framework application

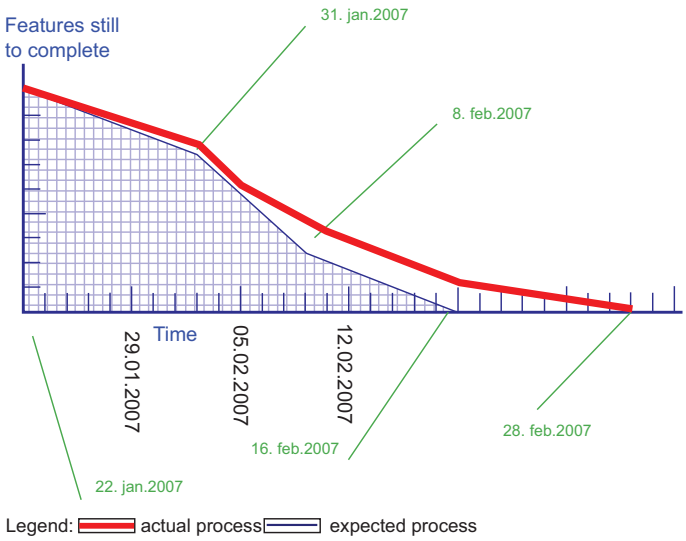

Fig. (9). Burn chart for the case study.

Documentation - only the following documents were produced: initial requirements specification , administrators guide, architecture and design document, and quality project plan. Tacit knowledge - agility is achieved with tacit-knowledge instead of documenting everything

Osmotic, Real-Time Face-To-Face Communication - Communication was picked up already by hearing others talk about something related to the project. Most of the time discussions were made in a room where all project participants were working

Feature Driven Development [32] - Tasks for development were not divided by stories or cards. Implementation tasks were divided by features of new software

Spiking, Walking Skeleton [17] - Before designing, a spike had to be made to be sure that it is possible. This was to eliminate technological risks

Simple Design - Incremental re-architecture. Design was developed step by step

Estimation using Comparison and Analogy - Estimates were done based on experience using comparison and analogy technique

Information Radiators - While discussing, drawing on design whiteboards was used

Access to Expert User - Team members had direct access to the customer *via* email

Frequent Delivery - SVN was used to keep compiling code. Always when a feature was implemented, it had to be implemented with the code in SVN. Repository usage offered the possibility to apply continuous integration, but then the automated unit tests would have been needed to complete or maximize the benefits of continuous integration. The automated tests were not in the scope of this project so continuous integration was implemented only at very high level each time somebody added something to the repository, one had to check compilation and test main features

Kick-Off Meeting - Kick-off meeting was held in the beginning of iterations

Status Reporting - After every week there were a status meeting. Daily stand-up meetings were unnecessary because project team was small and the information was always available to everybody

Code Reviews - There were two different activities: code review and code inspection. Code review is a brief inspection were the project manager divided source files between reviewers and the reviewers were looking at the general problems of source code (following the coding standard, monitoring, safety and names of classes, parameters and methods). Code inspection is applied when reviewers are looking more deeply into the source code functionality

Measurements of Quality Characteristics - The measurement activities are activities where somebody has to measure a quality characteristic specified in the quality plan. Whether there was a tool to measure a characteristic or not, the responsibilities were the same: first take the measurement; second write a short report about it to WIKI (when was the measurement taken, why was it taken, who took it, which method was used to measure a characteristic)

Burn Charts - Burn charts are effective way to present project status. Burn charts are graphical charts e.g. earned value vs. scheduled value [17]. This project used earned value graph in time (see Fig. **9**). As seen from the graph, on February 16th the project had not achieved its estimated goals. Some features were missing by that time. In this project developers could easily see the value that they had produced compared to the expected value. Features still to complete axis values were calculated using experience and analogy. All features got a relational weight and the total weight was the sum of all weights. After a feature completion , the total weight at that moment was increased and the corresponding line was represented in Fig. (**9**).

Table 5. Input Parameters for the Quantitative Models

Parameter	Unit	Project 1	Project 2
S	[SLOC]	10000	3020
N	[P]	4	3
c_{TR}	[m.u./P]	100	38.5
$t_{SIM-run}$	[H/run]	17	12
ir	[SLOC/meeting]	240	1510
c	[m.u./H]	100	38.5
$t_{D,I}$	[H/defect]	1	1
$n_{D,S}$	[defects]	1000	30
e_T	[%]	66.67	50
$t_{D,T}$	[H/defect]	10	10
E_D	[PH]	5088	360
r	[%]	5	10
n	[years]	4	4
T	[hours]	400	120
n_M	[meetings]	42	2
K	[PH/SLOC]	10.51	10.51

6). Case Study Conclusions

First, the case study used a hybrid software development method adapted to: i) project main characteristics, and ii) software development organization objectives of delivering high quality systems applying software quality standards and improved software development methods. It turned out that there were no major clashes applying the principles of ISO/IEC standards compared with the principles of agile manifesto. The only conflict area is the agile principle: responding to change following a plan, as ISO/IEC SQuaRE principles add some plan-driven characteristics to the agile foundations.

Second, a program was elaborated for applying an improved software engineering method. The guidelines for the program application are summarized as follows: creating an evaluation group ; creating a Quality Evaluation Project Plan; following the plan by appropriate quality activities.

Third, measurements, metrics, and models from Sections 4 and 5 that are inter-related in a systemic quantitative framework have been used. These allowed testing and validation of agility together with the ISO/IEC software quality standards in real life following the guidelines and achieving the expected quality goals.

Remarks:

a) Software quality metrics are measured with Research Standard Metrics (RSM) tool.

b) Quantitative economic yields of applying new (improved) software methods are calculated (estimated) using the value-based framework.

In the next paragraph we present the final results of applying the value-based framework.

6.2. Quantitative Framework Application

Metrics, models and formulas from Sections 4 and 5 are used to calculate quantitative yields of improving software engineering methods and therefore the quantitative yields of the improvement that will be made into the software development process. These formulas take into consideration the actual costs and benefits before (using an old software development process) and after a new improved software development process is applied and with some assumptions all economical indicators are calculated.

Before looking at the results, the assumptions made before applying the models, must be outlined. These formulas need different input parameters and different assumptions. First assumption is that the improvement made to software development process is similar to applying the Software Inspection Method (SIM). It must be mentioned that the guidelines of ISO/IEC SQuaRE quality standard are quite similar.

The input parameters for applying the quantitative (value-based) framework are summarized in Table **5** for Project 1 (illustrative example for our VF application), and Project 2 (case study).

The results obtained for the two projects by applying the models included in the value-based framework for managing software life-cycle when a new (improved) software engineering method is used are presented in Table **6**.

The main results from Table **6** can be summarized as follows: the return of investment (ROI); the gross benefit (GB) or the difference between life cycle costs before and after applying the new software engineering method - related to the number of defects that were actually found in the software; the net benefit (NB) by applying a new (improved)

Table 6. Quantitative Yields in the Software Life Cycle for Two Projects Using Improved(New) Software Engineering Methods

Metric	Unit	Project 1	Project 2
C_D	[m.u]	508 800	13 806
C_M	[m.u]	973 422	1100288
C_T	[m.u]	194 444	1150.5
C_I	[m.u]	70 833	920.4
C_{TR}	[m.u]	11240	115.05
C_{SIM}	[m.u]	1758740	1116280.18
TLC_b	[m.u]	4510000	1165867.23
TLC_a	[m.u]	1747500	1116165.63
GB	[m.u]	2762500	49701.6
AC	[m.u]	82073	1035.45
NB	[m.u]	2680426	48666.15
BCR		33.66	48
ROI	[%]	3266	4700
GB_D	[m.u]	2164491	33946.8
NB_D	[m.u]	2082417	32911.41
ROI_D	[%]	2537	3178
W_b	[SLOC/PH]	0.22	0.0994
W_a	[SLOC/PH]	0.57	0.1038
BEP	[m.u]	133991	24288.9
T_{BEP}	[H]	129	202.12

software engineering method; benefit to cost ratio (BCR); breakeven point (BEP) in units of cost [m.u.]; breakeven point in units of time TBEP [H].

7. CONCLUSIONS

Exploring the quantitative yields of software engineering methods in the software life cycle is a complex management problem that can be solved in a number of stages : i) designing or selecting an appropriate *software engineering method*; ii) elaborating a *program* for its application that is adequate for a particular software organization; iii) applying *measurements*, *metrics*, and *models* that are related in a *systemic framework* and iv) using a *proactive approach* to the achievement of the value-based results, instead of a reactive one.

We have theoretical and practical results applying hybrid software development method that combines agility and discipline using software quality management and ISO/IEC International Standard 25000 (SQuaRE)[29] on software product quality requirements and evaluation.

Our results on applying the above mentioned stages can be used as a basis for further experiments to gather empirical data in the area of exploring quantitative yields of software engineering methods and using the developed value framework for managing software life cycle.

ACKNOWLEDGEMENT

The authors would like to thank the reviewers and the Editor in Chief for their constructive suggestions and comments on a former version of this paper.

CONFLICT OF INTEREST

None Declared.

SOME ACRONYMS

VF	Value Framework
CM	Cost Models
WM	Productivity Models
QM	Quality Models
BM	Benefit Models
VM	Value Models
DF	Decision Framework
DDT	Dynamic Decision Trees
VOI	Value Of Information
NPV	Net Present Value
DO	Dynamic Optimization
NV	Net Value

SIM	Software Inspection Method
CMM	Capability Maturity Model
CMMI	Capability Maturity Model Integrated
COTS	Commercial Off The Shelf
XP	EXtreme Programming
SW	SoftWare
RUP	Rational Unified Process
ISO	International Organization for Standardization
IEC	International Electrotechnical Commission
UML	Unified Modeling Language
J2EE	Java 2 Enterprise Edition
XML	Extensible Markup Language
RSS	Rich Site Summary
OO	Object-Oriented
OMG	Object Management Group
WAP	Wireless Application Protocol
GPRS	General Packet Radio Service
SVN	Social Venture Network

REFERENCES

[1] A.J. Stoica, "Facets of Software Development Represented by Model Systems: Analysis and Enhancement," *14th International Forum on COCOMO/Software Cost Modeling, USC, CSE*, 1999.

[2] A.J. Stoica, "A Decisional Framework in Software Design," *International Conference on Software Engineering, ICSE 99, EDSER-1*, 1999.

[3] B. Boehm, *et al. "Software Cost Estimation with Cocomo II"*. Prentice Hall, 2000.

[4] I. Somerville, *Software Engineering, 9th ed.* Addison-Wesley, 2010.

[5] C. Kemerer, (Ed)., *Software Project Management: Readings and Cases*. Irwin, McGrew Hill, 1997.

[6] L. Franz, and J. Shih, "Estimating the Value of Inspections and Early Testing for Software Projects," *Hewlett-Packard Journal*, 1994.

[7] D. Rico, *ROI of Software Process Improvement*. Ross Publ., 2004.

[8] B. Boehm, and D. Port, "Conceptual Modeling Challenges for Model-Based Architecting and Software Engineering," *USC, CSE*, 1998.

[9] J. Fisher, "Model-Based System Engineering: A Paradigm," *INCOSE INSIGHT*, 1998, pp. 3-16.

[10] A. Gargaro, and A. Peterson, "Transitioning a Model-Based Software Engineering Architectural Style to Ada 95," *SEI Technical Report CMU/SEI, Tech. Rep. 96-TR-017*, 1996.

[11] Honeywell, *Model-Based Software Development*, Honeywell Technology Center, Minneapolis, 1998.

[12] OMG, "Model Driven Architecture," *http://www.omg.org/mda*, last updated on 05/31/2011.

[13] D. Garlan, and M. Shaw, "An Introduction to Software Architecture," *Advances in Software Engineering and Knowledge Engineering*, vol. 1, pp. 1-40, 1993.

[14] B. Boehm, and R. Ross, "Theory W software project management : principles and examples," *IEEE Transactions on Software Engineering*, vol. 15, no. 17, pp. 902-916, 1989.

[15] B. Boehm, "Get ready for agile methods, with care," *Computer*, 2002, pp. 64-69.

[16] S. Ambler, *Agile Modeling*. Wiley, 2002.

[17] A. Cockburn, *Agile Software Development*. Reading, MA : Addison-Wesley, 2002.

[18] J. Stapleton, *DSDM: Dynamic Systems Development Method*. Harlow: Addison-Wesley, 1997.

[19] S. Palmer, and J. Felsing, *A Practical Guide to Feature-Driven Development*. Englewood Cliffs, NJ: Prentice Hall, 2002.

[20] K. Schwaber, and M. Beedle, *Agile Software Development with Scrum*. Englewood Cliffs, NJ: Prentice Hall, 2001.

[21] K. Beck, and C. Andres, *Extreme Programming Explained: Embrace Change (2nd ed.)*. Addison-Wesley, 2004.

[22] J. Highsmith, *Adaptive Software Development*. Dorset House, 2000.

[23] P. Kruchten, *The Rational Unified Process (2nd ed.)* . Addison-Wesley, 2001.

[24] W. Humphrey, *Managing Technical People*. Addison-Wesley, 1997.

[25] W. Humphrey, *Introduction to Team Software Process*. Addison-Wesley, 2000.

[26] H. Mills, M. Dyer, and R. Linger, "Cleanroom Software Engineering," *IEEE Software*, vol. 4, no. 5, pp. 19-25, 1987.

[27] M. Paulk, C. Weber, B. Curtis, and M. Chrissis, *The Capability Maturity Model*. Addison-Wesley, 1994.

[28] ISO, "90003 Software Engineering: Guidelines for the Application of ISO 9001:2000 to Computer Software," 2004.

[29] ISO, "25000: Software Engineering–Software Product Quality Requirements and Evaluation (SQuaRE)–Guide to SQuaRE," 2006.

[30] ISO, "9001 Quality Systems - Model for Quality Assurance in Design/Development, Production, Installation, and Servicing," 1994.

[31] T. DeMarco, and B. Boehm, "The agile methods fray," *IEEE Computer*, vol. 35, no. 6, pp. 90-92, 2002.

[32] B. Boehm and R. Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley, 2003.

[33] B. Boehm, "Some Future Trends and Implications for Systems and Software Engineering Processes," *Systems Engineering, Wiley Periodicals Inc*, vol. 9, no. 1, pp. 1-19, 2006.

[34] A.J. Stoica, and M. Nael, "Agile Software Development and ISO/IEC Software Quality Standards : Measuring Economic Benefits and Calculating Quantitative Yields," *25th International Forum on Software and System Cost Modeling, USC, CSE*, 2010.

Received: April 20, 2011 Revised: June 26, 2011 Accepted: July 7, 2011