OptimalSQM:Integrated and Optimized Software Quality Management

Ljubomir Lazić
Technical Faculty, State University of Novi Pazar
Vuka Karadžića bb, 36 300 Novi Pazar,
SERBIA
llazic@np.ac.yu, http://www.np.ac.yu

Nikos E Mastorakis
Technical University of Sofia,
English Language Faculty of Engineering
Industrial Engineering
Sofia 1000
Sofia
BULGARIA
http://www.wseas.org/mastorakis

Abstract: - Software testing provides a means to reduce errors, cut maintenance and overall software costs. Early in the history of software development, testing was confined to testing the finished code, but, testing is more of a quality control mechanism. However, as the practice of software development has evolved, there has been increasing interest in expanding the role of testing upwards in the SDLC stages, embedding testing throughout the systems development process. Numerous software development and testing methodologies, tools, and techniques have emerged over the last few decades promising to enhance software quality. While it can be argued that there has been some improvement it is apparent that many of the techniques and tools are isolated to a specific lifecycle phase or functional area. This paper presents a set of best practice models and techniques integrated in optimized and quantitatively managed software testing process (OptimalSQM), expanding testing throughout the SDLC. Further, we explained how can Quantitative Defect Management (QDM) Model be enhanced to be practically useful for determining which activities need to be addressed to improve the degree of early and cost-effective software fault detection with assured confidence, then optimality and stability criteria of very complex STP dynamics problem control is proposed.

Key-Words: - Software Testing, Quality, Testing optimization, Cost of Quality, Testing stability criteria

1 Introduction

The software development industry spends more than half of its budget on maintenance related activities. Software testing provides a means to reduce errors, cut maintenance and overall software costs. The importance of software testing has been emphasized more and more, as the quality of software affects its benefit to companies significantly [1-4]. This paper presents some research results of ongoing project [5-7][1], designed to study software defect data as a means toward identifying where resources should be allocated most effectively to provide the highest quality of software product while reducing the overall cost of software testing. The identification and removal of software defects constitutes the basis of the software testing process a fact which inevitably places increased emphasis on defect related software measurements. Early in the history of software development, testing was confined to testing the finished code, but, testing is more of a quality control mechanism. However, as the practice of software development has evolved, there has been

[1] This work has been done within the project 'Integrated and Optimized Software Testing and Maintenance Process', supported in part by the Ministry of Science and Technological Development of the Republic of Serbia under Grant No. TR-13018.

increasing interest in expanding the role of testing upwards in the SDLC stages, embedding testing throughout the systems development process, so, testing becomes a parallel process. Avoidable rework consumes a large part of development projects, i.e. 20-80 percent depending on the maturity of the organization and the complexity of the products [9]. High amounts of avoidable rework commonly occur when having many faults left to correct in late stages of a project. In fact, research studies indicate that the cost of rework could be decreased by up to 50 percent by finding more faults earlier [2, 5, 9]. Numerous software development and testing methodologies, tools, and techniques have emerged over the last few decades promising to enhance software quality. While it can be argued that there has been some improvement it is apparent that many of the techniques and tools are isolated to a specific lifecycle phase or functional area.

This paper focuses on software testing and the measurements which allow for the quantitative evaluation of this critical software development process. The software testing process requires practical measurements for the quantification of all software testing phases. Software product quality and software testing process (STP) improvement commence with addressing the testing process in a quantitative manner [7]. The continuous monitoring of the testing process allows for establishing an adequate level of confidence for the release of software products and for the quantification of software risks, elements which traditionally have plagued the software industry. The mechanism for this study is development of a series of simulation models to improve STP [7,8].

The first phase of model development is presented in this paper. Ongoing work will involve extensive data collection regarding business processes followed by the use of simulation in the development of decision models [8,9]. In this paper, Quality and Efficiency in Software Testing by our *OptimalSQM* framework is described and its components defined and exemplified. It also discusses practical applications of *OptimalSQM* and research model for investigating its antecedents and impacts is presented. *OptimalSQM* provide alignment between testing and development which has been raised as an issue for successful systems development. Missing however have been actionable how to methodologies for assessing and enhancing such alignment [12,13,16]. This paper attempts to fill this gap by describing a systematic methodology, a development-testing alignment (DTA) methodology which posits that such alignment leads to beneficial effects such as lower costs and shorter time of development, greater system quality, fewer errors and a better relationship between the corporate IT unit and customers in business functions who have commissioned new systems. This methodology considers alignment at both strategy and execution levels. By dissecting alignment into internal (within) and external (between) categories, it outlines pragmatic mechanisms by which the coherence between the internal components of developer-tester alignment can be assessed and managed. Further, we explained how can Quantitative Defect Management (QDM) Model be enhanced to be practically useful for determining which activities need to be addressed to improve the degree of early and cost-effective software fault detection with assured confidence, then optimality and stability criteria of very complex STP dynamics problem control is proposed.

2 Need for Research

Cost to an organization (both in dollars and in image) is significant when software defects are identified after installation at a client site. Our project intends to identify areas where improvements in software testing resource allocations would provide added value to the organization. This paper proposes a development-testing alignment (DTA) methodology which posits that such alignment leads to beneficial effects such as lower costs and shorter time of development, greater system quality, fewer errors and a better relationship between the corporate IT unit and customers in business functions who have commissioned new systems (see Fig. 2 and 3 below). Alignment models and measurements have been studied in other related contexts [16] but never within corporate IT units and specifically between the development and testing functions. The paper therefore decomposes DT alignment into a series of aspects for the purpose of assessing and analyzing each of the construct. These aspects are drawn from the overarching framework developed initially from prior literature [8,16]. The DTA methodology will allow IT managers to improve the effectiveness of testing and development by both synergistically integrating testing in the development process and by aligning the testing and development units in terms of strategy and execution capability.

Organiztions that engage in software development and testing benefit significantly if their management team has tools to assist them in determining the most effective use of financial resources that might result in the fewest software errors in delivered systems [2-10,22-26]. To be most

effective, this tool needs to be developed after a thorough review of the specific organization's testing data [17,26]. Once developed, the tool will identify the specific phases and processes during the evelopment life cycle where additional resources would provide the best return on investment and highest software quality.

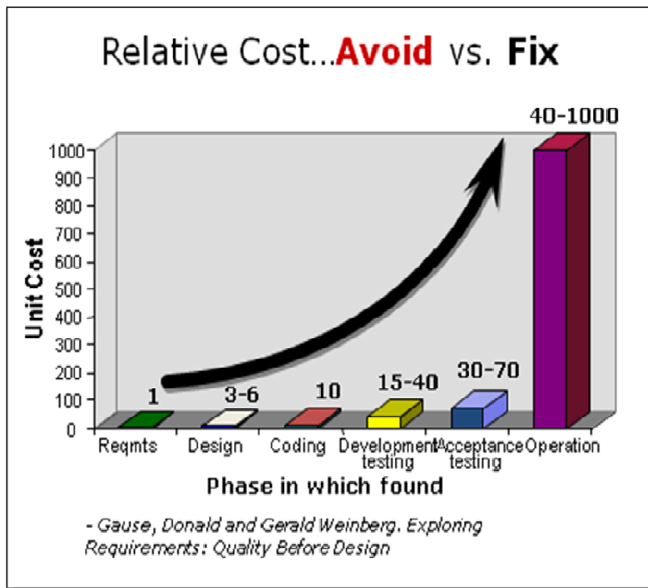

Fig. 1 Average Cost Of Defect Removal [2]

The use of this tool will provide a major reduction in the number and severity of software defects that exist after software testing. It will also reduce the overall cost of software testing by focusing on the appropriate process for a specific organizational environment [7,9,17-19,26]. To summarize, the purpose of this research is to increase software quality and reduce overall costs of software testing by focusing resources where they provide the most value. According to Gartner [14], on average, 7% of software functionality that was paid for is actually used, while 85% of IT projects failing to meet objectives (32% being cancelled outright). Dhaliwal and Onita [13] posit that many of these development failures are a result of poorly executed development process. These employ either inadequate development models or flawed implementation due, in part, to the lack of proper testing and effective collaborative mechanisms between testers and developers. These issues have not yet found a proper solution, due, in part, to a lack of a methodology that would allow the analysis and correction of software development processes. A review of the testing and development literature reveals that relations between the development and testing functions are lacking for projects of medium and large magnitude, where testing is separate from the development activities [12,15].

2.1 Research Questions

Based on the outcome of the evaluation of related work conducted in the previous subsection, the our project has identified some challenges to address. The challenges can be broken down into five sub-questions to address in this paper. During the work on this project[1] several research questions were formulated which the research then was based upon. The initial main research question that was posed for the complete research in this project was:

RQ1 or Main Research Question: How can software testing be performed efficiently and effectively i.e. **Optimal**, that is, do we have a framework model targeted specific software testing domains or problem classes described below in **RQ2** to **RQ5**?

To be able to address the main research question several other research questions needed to be answered first (**RQ2–RQ5**). The first question that needed an answer, after the main research question was formulated, was:

RQ2: Which metric or set of metrics can assess effectiveness of test detectin techniques and what is the potential in combining different software testing techniques with respect to effectiveness (and to some extent efficiency)?

Thus, since this project is based upon the main research question, it was worthwhile taking the time to examine the current practice in different projects and see how software quality is measured and, especially, software testing was practiced [1-8]. The answer to this research question is to be found in Section 3 and 4 together with an analysis of how software testing is used in different types of projects. To put it short, the answer to **RQ2** divided the research, as presented in this paper, into two areas covering effectiveness in software testing techniques and efficiency in software testing with development-testing alignment (DTA) methodology [5-9] which posits that such alignment leads to beneficial effects such as lower costs and shorter time of development, greater system quality, fewer errors and a better relationship between the corporate IT unit and customers in business functions who have commissioned new systems. To begin with, the research aimed at exploring the factor of defect detection and removing effectiveness (DRE) during SDLC (**RQ3**) while

later focusing on early aspects of software cost of quality. In order to examine if the current practice in software development projects was satisfactory for developing software with sufficient quality and budget constraint, **RQ3** evolved into:

RQ3: Which metric or set of metrics can identify and prioritize software quality attributes, can assess cost of software quality management process in a specific project i.e. how to optimize software quality to pay off investment in STP improvement (ROI)?

Simply put, the main research question might have been a question of finding optimization model of software Quality and Efficiency in Software Testing by an *OptimalSQM* framework and its components defined in advance at start point of SDLC. The *OptimalSQM* framework needs a systematic model which enables to minimize the cost of switching between test plan alternatives, when the current choice cannot fulfill the quality constraints, to enable software designers to achieve a higher quality for their design, a better insight into quality predictions for their design choices that evolved to the **RQ4**:

RQ4: Which metric or set of metrics can identify and prioritize improvements to achieve early and cost-effective software fault detection, can assess the improvement potential of improving the degree of early and cost-effective software fault detection?

The answer to **RQ4** can be found in Section 5 which introduces new kinds of STP improvenent and hence indirectly led to Research Question 5:

RQ5: How should a software development organization apply the metric(s) suggested above for assessing ongoing and finished projects with an **Dynamic Control Model** view i.e. What are **optimality** and **stability** criteria of very complex STP dynamics problem control?

The answer to **RQ5** can be found in Section 6 and 7. Section 6 explain how can Quantitative Defect Management (QDM) Model be enhanced (as answer to **RQ4**) to be practically useful for determining which activities need to be addressed to improve the degree of early and cost-effective software fault detection with assured confidence, **optimality** and **stability** criteria of very complex STP dynamics problem control (described in Section 7).

Before any work on solving a particular research questions starts (a research question is basically a formalization of a particular problem that needs to be solved) a researcher needs to look at how the problem should be solved. To be able to do this, one must choose a research methodology. Iterative approaches for improvement exist in the quality management area. The PDCA (plando-check-act) or "Shewhart Cycle", the WV (or zigzag) framework and the DMAIC (definemeasure-analyze-improve-control) cycles are analogous methods to capture a generic framework for the improvement of a process or system [1,3,8]. A similar model, the "simulate-test-evaluate process" iterative experimentation cycle was developed by the office of the US Secretary of Defense, called the Simulation, Test and Evaluation Process (DoD STEP framework) to integrate M&S into the test and evaluation process of the system/software under test (SUT) [17]. A basic rule from cybernetics - that a long time lag between the output signal from the controlled system and feedback to the controller causes instability in the system - applies to SDP-STP processes as well. Long design iteration loops with late feedback drive cost and schedule overruns in SDP-STP.

3 Key Concepts of Developer-Tester Alignment solution in integrated, quantitatively managed and optimized software testing process

When design and testing activities are not coupled, the information testing provides on product design is delivered at a wrong point in the process. This late information is either not useful any more or shows design problems too late, causing undesired late rework. Thus, iteration cycles should be kept short and rapid. However, this is difficult in the context of a number of interrelated activities without a model to facilitate process analysis and improvement.

To address the research questions stated above, multiple studies have been conducted [5-8] about alignment between the development and testing functions which can be defined as the strategic and operational fit between the development and testing functions on components of strategy and capabilities [13-16]. Since systems development as well as systems testing are integral parts of the corporate technology acquisition strategy, they too have to be aligned to ensure business success. In many organizations, there is a gap, or misalignment, at the strategic and/or execution level, between the development and testing groups as well as between individual testers and developers. To correct these misalignments, this paper proposes a methodology, grouped under the DTA model [13] that draws upon

the strategic alignment model initially proposed in [16]. This DTA model focuses on the fit between the development and testing functions. A key goal of this research is to develop a methodology for applying these concepts within the corporate IT unit tasked with building and implementing business system applications. A high level of integration of business and IT plans facilitates communication and collaboration [16]. Similarly, in the areas of development and testing, a high level of integration and correspondence at both the execution and strategic levels may also facilitate communication and collaboration. Integration represents the level of linkage between development and testing, while correspondence represents how closely their capabilities mirror and complement each other. Varying levels of alignment can either promote or hinder integration and correspondence. This is a common characteristic of all alignment models in the literature as verified by Dhaliwal, J. and Onita C. in their work [13]. Figure 2 details the key structural and flow components of the DT alignment model for development and testing within the corporate IT unit. This model decomposes the alignment of the development and testing functions along three key flow dimensions: 1) strategic alignment, 2) capabilities alignment, and 3) strategy-execution alignment.

The first structural component, development strategy looks at strategic choices of the development function. This component is comprised of three key aspects: the scope of development, governance of development and development resources. Here the scope of IT development is defined in terms of IT goals that support the business strategy.

The second structural component, development capabilities, has three key aspects: development process, development skills and development architecture. These directly impact the applications being developed, the tools used in development processes, as well as the models or frameworks employed in the development process. Decisions about development models, such as SDLC, RAD, prototyping, etc., the skills and competencies of the development personnel are also considered at this level. On the testing side, the third structural component, testing strategy focuses on three key aspects: the scope of testing, issues regarding responsibilities and resources and the governance and reporting structure of the testing function. The fourth structural component describes the testing capabilities and has three aspects, testing processes, testing skill/competencies and testing architecture. The specific methods of testing (traditional, V-mode, iterative), as well as choices about testing tools, architecture, communication structure, etc. are considered and analyzed from an alignment perspective.

The individual skills of testing personnel are also assessed. In conclusion, the top two quadrants of Figure 2 represent the strategy level while the lower two quadrants represent the capability level. The left two quadrants represent the development function while the right two quadrants represent the testing function.

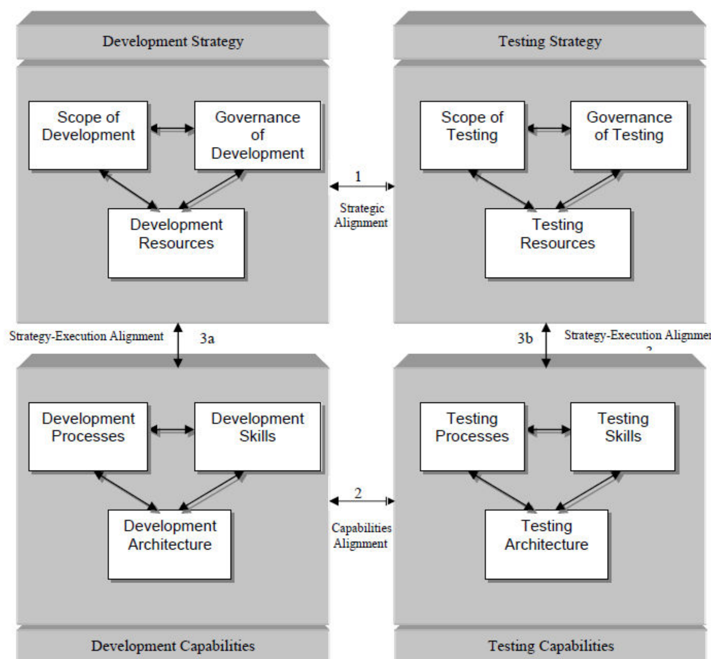

Fig. 2 Alignment model for testing and development (adapted from [13])

DT alignment implies that all four dimensions are matched in capabilities, resources, structure, etc. This does not mean that they have to be similar, but that testing complements development and acts as an enabler of development success by providing verification, validation and bug-finding services. Each structural component (quadrant) of the Alignment Model deals with alignment from a double perspective: strategy/capabilities and development/testing. DT Alignment also has three flow dimensions (as represented by the numbered vertical and horizontal arrows): strategic alignment, capabilities alignment, and strategy-execution alignment.

3.1 The testing Strategy-Execution Alignment

The testing Strategy-Execution Alignment (arrow 3b) deals with the ability of testing capabilities

(competencies, tools and methodologies) to support the execution of stated testing strategies. Testing strategies have to be executable, and testing capabilities have to empower and support the strategic goals and decisions. Prior studies have identified additional influences that impact the alignment between the components of the DTA framework. Similarly, when talking about alignment between testing and development, shared domain knowledge of development executives and testing executives will, most likely, positively influence the level of alignment between the two functions. When testing executives have development experience and/or knowledge, and when development executives have testing experience and/or knowledge, their decisions will lead to better alignment of the functions. This is especially true if participative decision making takes place, where testing executives are part of the development functions decision making and development executives are part of the testing function's decision making process.

Improving any process can be facilitated by proper planning and by following detailed and fitting methodologies and techniques. Based on case study and field study approaches [8], this study proposes a methodology for achieving DT Alignment (see Fig. 3) through Collaborative Techniques & Technology which Enables *OptimalSQM* to be realised. The methodology is derived from a survey of the literature from Strategic Alignment [13-16] Testing [1-4], [8-10] to Project Management and Information Systems development methods [10-16]. To improve the reliability and validity of this methodology, alignment case studies and field studies were conducted and real life examples are given to improve the applicability of the methodology. A list of techniques is also mapped onto each step of the methodology. While not exhaustive, this toolbox gives IT managers a good idea about the available techniques that can used when attempting to secure high levels of DT Alignment/Realignment strategy.

3.2 Integrated, quantitatively managed and optimized software testing process - OptimalSQM solution

To answer the main research question (**RQ1**) we applied DTA model, desribed above, in *OptimalSQM* framework which combine best practice from Design of Experiments, Modeling & Simulation, integrated practical software measurement, Six Sigma strategy, Earned (Economic) Value Management (EVM) and Risk Management (RM) methodology through simulation-based software testing scenarios at various abstraction levels of the software under test (SUT) to manage stable (predictable and controllable) software testing process at lowest risk, at an affordable price and time [8,9], [17,18] as depicted in Fig. 4. Unlike conventional approaches to software testing (e.g. structural and functional testing) which are applied to the software under test without an explicit optimization goal, the *IOSTP* with embedded **R**isk **B**ased **O**ptimized **STP** (*RBOSTP*) approach designs an optimal testing strategy to achieve an explicit optimization goal, given a priori [8,17].

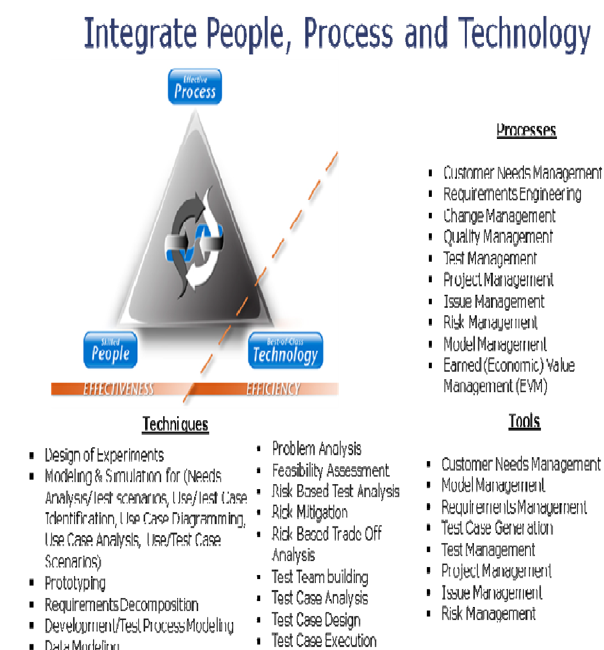

Fig. 3 Collaborative Techniques & Technology Enables *OptimalSQM* realisation

This leads to an adaptive software testing strategy. A non-adaptive software testing strategy specifies what test suite or what next test case should be generated, e.g. random testing methods, whereas an adaptive software testing strategy specifies what testing policy should be employed next and thus, in turn, what test suite or test case should be generated next in accordance with the new testing policy to maximize test activity efficacy and efficiency subject to time-schedule and budget constraints. The process is based on a foundation of operations research, experimental design, mathematical optimization, statistical analyses, as well as validation, verification, and accreditation techniques.

The use of state-of-the-art methods and tools for planning, information, management, design, cost trade-off analysis, and modeling and simulation, Six Sigma strategy significantly improves STP effectiveness as in Fig. 4 which graphically illustrates a generic **IOSTP** framework that makes core of the *OptimalSQM* framework [8].
The main components of IOSTP with embedded RBOSTP approach to STP:

- Integrate testing into the entire development process
- Implement test planning early in the life cycle via Simulation based assessment of test scenarios
- Automate testing, where practical to increase testing efficiency
- Measure and manage testing process to maximize risk reduction
- Exploit Design of Experiments techniques (optimized design plans, Orthogonal Arrays etc.)
- Apply Modeling and Simulation combined with Prototyping
- Continually improve testing process by pro-active, preventive (failure mode analysis) Six Sigma DMAIC model
- Continually monitor Cost-Performance Trade-Offs (Risk-based Optimization model, Economic Value and ROI driven STP).

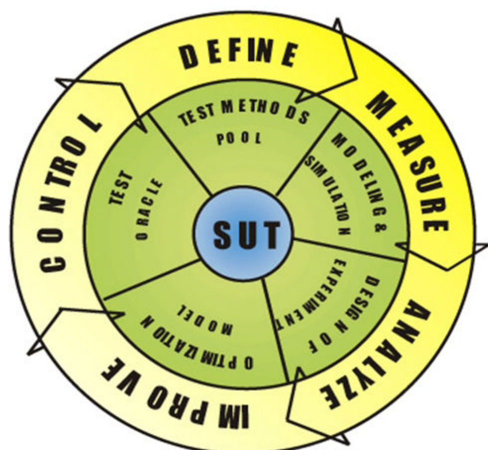

Fig. 4 Integrated and optimized software testing process (IOSTP) framework, a core of *OptimalSQM* framework [8]

Framework models are similar to the structural view, but their primary emphasis is on the (usually singular) coherent structure of the whole system, as opposed to concentrating on its composition. IOSTP framework model targeted specific software testing domains or problem classes described above. IOSTP is a systematic approach to product development (acquisition) which increases customer satisfaction through a timely collaboration of necessary disciplines throughout the life cycle. Successful definition and implementation of IOSTP can result in:

- Reduced Cycle Time to Deliver a Product
- Reduced System and Product Costs
- Reduced Risk

In order to significantly improve software testing efficiency and effectiveness for the detection and removal of requirements and design defects in our framework of IOSTP, during 3 years of the IOSTP framework deployment to STP of embedded-software critical system such as Automated Target Tracking Radar System (ATTRS) [17], we calculated overall value returned on each dollar invested i.e. ROI of 100:1 .

4 Optimum DDTs combination selection and optimization study in OptimalSQM

Answer to the research queation - **RQ2** divided the research, as presented in this paper, into two areas: (1) covering effectiveness in software testing techniques (defect detection techniques – DDT), and (2) efficiency in software testing with development-testing alignment (DTA) methodology is given in our works [5-8] which posits that such alignment leads to beneficial effects such as lower costs and shorter time of development, greater system quality, fewer errors and a better relationship between the corporate IT unit and customers in business functions who have commissioned new systems.

The central elements of IOSTP with embedded RBOSTP are finding optimal DDTs combination choices for every software development phase that maximize all over Defect Detection and Removement Effectiveness in OptimalSQM: the acquisition of information that is credible; avoiding duplication throughout the life cycle; and the reuse of data, tools, and information. Among numerous defect detection techniques choices we reduced their number using Borda voting method to rank DDT candidates from most powerful on the basis of multiple evaluation criteria we have been established [6]. Using OART novel approach, optimum combination of software defect detection techniques choices for every software development phase that maximize overall Defect Detection Effectiveness of STP is determined. IOSTP framework combines few engineering and scientific areas such as: Design of Experiments, Modeling &

Simulation, integrated practical software measurement, Six Sigma strategy, Earned (Economic) Value Management (EVM) and Risk Management (RM) methodology through simulation-based software testing scenarios at various abstraction levels of the SUT to manage stable (predictable and controllable) software testing process at lowest risk, at an affordable price and time [6].

Our study [6] focuses on rapid multidisciplinary analysis and evaluation-on-a-DRE maximum-basis for DDT combination choices selection for each test phase activities in an traditional SDP i.e. P1- software requirement, P2- High level design, P3- Low Level Design, P4- code under test, P5- integration test, P6- system under test and finally P7- Acceptance test, recall section 5. Different Defect Detection Strategy and Techniques options, together with critical STP variables performance characteristics (e.g. DRE, cost, duration), are studied to optimize design, development, test and evaluation (DDT&E) cost using orthogonal arrays for computer experiments [8,9,17]. This paper presents a novel OACE (Orthogonal Arrays for Computet Experiment) approach for software testing process (STP) optimization study finding optimum combination of software defect detection techniques (DDT) choices for every software development phase that maximize all over Defect Removal Effectiveness (DRE) of STP. The optimum combination of software defect detection techniques choices were determined applying orthogonal arrays constructed for post mortem designed experiment with collected defect data of a real project [6]. First, we applied adapted Borda voting method, on similar way, to rank all used Defect Detection Techniques (DDT) through software development life cycle from most-to-least performance and quality characteristics of DDT in revealing software faults (bugs, errors). In this way we reduced huge possible number of DDTs, in particular, the DDT with the highest Borda Count is the best DDT according to testers Performance and Quality multi-criteria assessment, the DDT with the second highest count is the next DDT with highest score, and so forth to only three most ranked DDT. According to testers assessment of 5 most frequently used DDT in IOSTP [6,8]: DDT1= Inspection – DBR, DDT2= PBR, DDT3= CEG+BOR+MI, DDT4= M&S, DDT5= Hybrid (Category Partition, Boundary value analysis, Path testing etc.) three of DDTs have the highest rank 0 i.e. DDT1=DDT2=DDT4=0, then DDT3= CEG+BOR+MI is next ranked and the last was DDT5. Because of that we will group those three DDT with highest rank 0, call them Static Test Techniques – TT1 and treat all three DDTs as one factor in optimization experiment applying Orthogonal Arrays as Optimization Strategy. Next high Borda ranked DDT4= CEG+BOR+MI we designate with TT2 and the last ranked DDT5 as TT3.

In this study, design of maximum DRE percentage of STP optimization problem solving with best DDT choice combination in each phase P1 to P7 as controlled variables values is determined by designed experiment plan using orthogonal arrays designed for this computer experiment (OACE). To simplify the analysis such as decreasing factor's values (only three DDT number) applying Borda Ranking of DDT candidates with highest rank, several design disciplines were decoupled from the present analysis. Seven major test phases P1 to P7 for accounting maximum DRE percentage all over STP fault injection and removal model (see Fig. 8, 17 and 18 below) for DDT candidate selection in each test phase were determined. These were the Static Test Techniques – TT1 (consisting of three DDTs as one factor in optimization experiment applying Orthogonal Arrays as Optimization Strategy), the TT2 i.e. DDT4= CEG+BOR+MI and TT3 – Hybrid Detection Technique= DDT5 (consisting of Category Partition, Boundary value analysis, Path testing etc.). The objective of this investigation was then to determine the best combination of Test Techniques (TTi , i=1,2 and 3) options for the seven major test phase activities sections optimized for STD&STP maximum DRE percentage under cost and time constraints according to IOSTP benefit index maximization in (1) [8,17].

As the next step, least squares regression analysis is used to fit the second order approximation model given by equation (1) to the DRE data in terms of the seven design variables P_i, i=1 to 7. This parametric model accounts for the response surface curvature (square terms) and two factor interactions (cross terms) i.e. RSM:

$$DRE(\%) = 111.71 - 2.58 * P1 + 1.22 * P2 - 1.95 * P3 - 7.61 * P4 - 0.69 * P5 + 0.94 * P6 - 13.04 * P7 - 0.36 * P2^2 + 1.46 * P4^2 + 0.79 * P5^2 - 0.36 P6^2 + 3.15 * P7^2 \quad (1)$$

Note that, in this response surface approximation model, the parameter values for P_i design variables are restricted to 1 (TT1), or 2 (TT2), or 3 (TT3). In Table 1, a Maximum DRE (%) value and corresponding Test Techniques choices (TT1,TT2 and TT2) per test phase solution is given.

Table 1 Maximum DRE (%) value and corresponding Test Techniques choices per test phase solution

P1	P2	P3	P4	P5	P6	P7	DRE (%)
TT1	*TT2*	*TT1*	*TT1*	*TT3*	*TT2*	*TT2*	*94.03*

At these levels, the IOSTP DRE was predicted to be 94.03 % using a second order prediction model (1). As a next step, a verification analysis was performed. The DRE (%) of an IOSTP calculated from these test techniques choices, according to the post-mortem real project data using optimized DDT chices from Table 1, we computed DRE (%) to be 93.43 % . Difference is 0.6%=94.03%-93.43% that is acceptable to validate our prediction model for DRE (%) in equation (1) for optimal DDT combination choice given in Table 1.
Optimal combination of DDT choices per phase P given in Table 1 made increase of about 6 %, compared to un-optimized DDTs combination per each test phase we used in our real project in which we achieved DRE of 87.43 % in our case study.

5 Software qulity economics

From a developer's perspective, there are two types of benefits that can accrue from the implementation of good software quality practices and tools: money and time. The investment in software quality, particularly in software testing, like any investment has an immediate cost, with an expected net payback. There is where Quality Cost Analysis could be used as effective tool to make them understand the ROI. In this section, we will define techniques to, analyze and interpret return on the testing investment (ROTI) values: Financial ROI and Schedule Benefits as one answer to **RQ3** based on our studies [5,18,19] i.e. Which metric or set of metrics can identify and prioritize software quality attributes, can assess cost of software quality management process in a specific project i.e. how to optimize software quality?

In our work [19] we proposed a model which traces design decisions and the possible alternatives. With this model it is possible to minimize the cost of switching between design alternatives, when the current choice cannot fulfill the quality constraints. With this model we do not aim to automate the software design process or the identification of design alternatives. Much rather we aim to define a method with which it is possible to assist the software engineer in evaluating design alternatives and adjusting design decisions in a systematic manner.

There are some prepositions, which are not being tested comprehensively, but some useful Economic Model of Software Quality Costs and data from industry are described in this article [1,3,4]. Significant research is needed to understand the economics of implementing quality practices and its behaviour. Such research must evaluate the cost benefit trade-offs in investing in quality practices where the returns are maximized over the software development life cycle.
The total of the quality costs includes prevention costs of nonconformance to requirements, appraising costs of product or service
for conformance to requirements, and failure costs of products not meeting requirements. As the quality function evolved from inspection (quality control) to more preventive activities (quality assurance), quality cost collection was expanded into prevention, appraisal, and failure costs. Failure costs are divided into two subcategories: internal and external. Dan Houston [23] has defined Cost of quality in his article "Cost of Software Quality: A Means of Promoting Software Process Improvement" as follows;

$$CoSQ = Prevention_{Cost} + Appraisal_{Cost} + Internal\ failure_{Cost} + External\ failure_{Cost}$$

By now we have clear understanding of four components of the Quality cost. With the help of these four components we will discuss the theoretical model suggested by researcher based on the results gathered from the manufacturing industries. Following Fig. 5, is graphical presentation of the CoSQ given by most researchers [3-4], [19-23]. The graph below is showing that for achieving high reliability, close to red dot (almost zero defect) the cost is very high but achieving a reasonable level (area between two green dots) of quality does not require very high cost. To remove defect after reaching at very low defect density the cost of detection would be very high (Rs.500/KLOC) whereas the defect detection was relatively easy as numbers of defect were high (high defect density) the cost to remove defect is approximately 10 times lesser.

Cost mentioned on the graph are imaginary numbers just to give an idea that cost of defect removal at high defect density would be lower and cost at low defect density would be high. Several studies [19-23] described meanings of these quality cost categories as follows:

• Prevention costs (PC) are those costs associated with quality planning, designing, implementing and managing the quality system, auditing the system, supplier surveys, and process improvements.

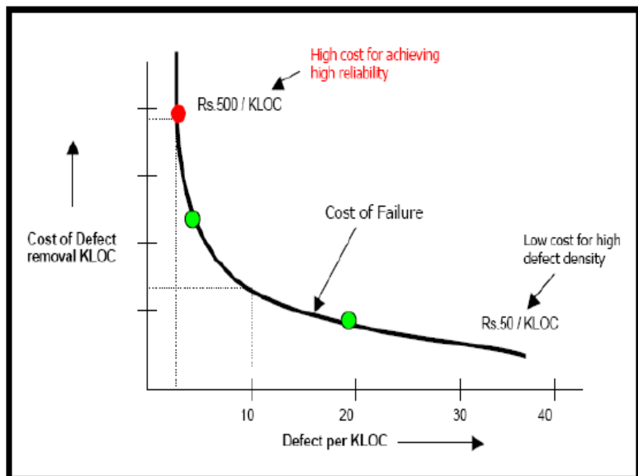

Fig. 5. The cost of high reliability

• Appraisal costs (AC) are associated with measuring, evaluating, or auditing products, and product materials to ensure conformance with quality standards and performance requirements.
• Failure costs (FC) are those losses associated with the production of a nonconforming product; they can be divided into internal and external.
• Internal failure costs (IFC) are associated with failures and defects of processes, equipment, products, and product materials that fail to meet quality standards or requirements.
• External failure costs (EFC) are generated by defective products, services, and processes during customer use. They include warranties, complaints, replacements or recalls, repairs, poor packaging, handling, and customer returns.

5.1 Cost of Software Quality (CoSQ)

The costs of achieving quality and the costs due to lack of quality have an inverse relationship to one another: as the investment in achieving quality increases, the costs due to lack of quality decrease. This theoretical model is shown below in Fig. 6. This shows that as appraisal and prevention cost increases, the failure cost will decrease until an optimum point is reached. After this optimum point, the increase in appraisal will not be offset by the decreased in failure cost. Researcher have noticed that in the initial phase appraisal measures cause internal failure to increase as these measures detect more errors at early stages, but error removal at early stage is much cheaper compare to error removal at later stage. But overall appraisal activities decrease external failure as a result total failure decreases. A small increase in prevention measures will normally create a major decrease in total quality cost.

Cost of quality represents any and all costs that organization incurs from having to repeat a process more than once in order to complete the work correctly. Cost of software Quality (CoSQ) is useful to enable our understanding of the economic trade-offs involved in delivering good-quality software. Commonly used in manufacturing, its adaptation to software offers the promise of preventing poor quality but, unfortunately, has seen little use to date. Different authors and researcher have used different ways to classify components for quality cost [5-9], if we look carefully their understanding about various components are approximately the same.

5.2 Statement Of the Problem

A key metric for measuring and benchmarking the software testing efficacy is by measuring the percentage of possible defects removed from the product at any point in time. Both a project and process metric – can measure effectiveness of quality activities or the quality of a all over project by:

$$DRE = E/(E+D) \tag{2}$$

Where E is the number of errors found before delivery to the end user, and D is the number of errors found after delivery.

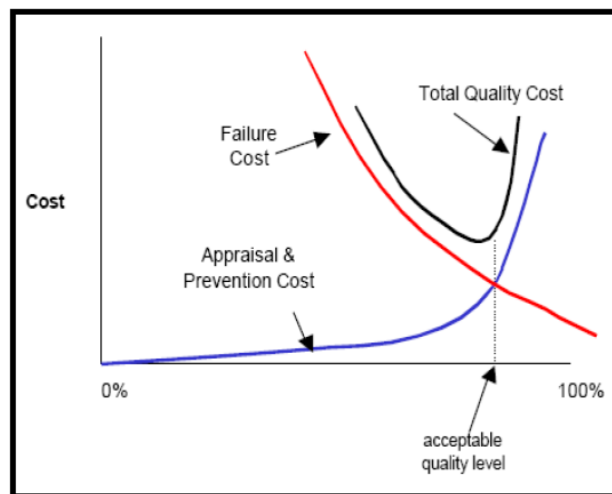

Fig. 6 Model of software quality cost

The goal is to have *DRE* close to 100%. The same approach is applied to every test phase denoted wit *i* as shown on Fig. 7:

$$DRE_i = E_i / (E_i + E_{i+1}) \qquad (3)$$

Where E_i is the number of errors found in a software engineering activity *i*, and E_{i+1} is the number of errors that were traceable to errors that were not discovered in software engineering activity *i*. The goal is to have this DRE_i approach to 100% as well i.e., errors are filtered out before they reach the next activity. Projects that use the same team and the same development processes can reasonably expect that the *DRE* from one project to the next are similar. For example, if on the previous project, you removed 80% of the possible requirements defects using inspections, then you can expect to remove ~80% on the next project. Or if you know that your historical data shows that you typically remove 90% before shipment, and for this project, you've used the same process, met the same kind of release criteria, and have found 400 defects so far, then there probably are ~50 defects that you will find after you release. How to combine Defect Detection Technique (DDT) to achieve high *DRE*, let say >85%, as a threshold for IOSTP required effectiveness [2-5], is explained in previous Section 4, which describe optimum combination of software defect detection techniques choices. Note that the defects discussed in this section include all severity levels, ranging from severity 1: *activity stoppers*, down to severity 4. Obviously, it is important to measure defect severity levels as well as recording numbers of defects.

5.3 The Real Cost Of Software Defects

It is obvious that the longer a defective application evolves the more costly it is to repair. But how much more? The answer might surprise you. According to the collected metrics of one software development organization, a bug that costs $1 to fix on the programmer's desktop costs $100 to fix once it is incorporated into a complete program, and many thousands of dollars if it is identified after the software has been deployed in the field [10], as described on Fig. 8. Barry Boehm, one of the industry's leading experts on software quality, has published several studies [11] over nearly three decades that demonstrate how the cost for removing a software defect grows exponentially for each downstream phase of the development lifecycle in which it remains undiscovered. Since the original study, Boehm's results have been confirmed in a number of subsequent studies [5-7]. Further, another major research project conducted recently by the United States Department of Commerce, National Institute of Standards and Technology (NIST) showed that in a typical software development project, fully 80% of software development dollars are spent correcting software defects.

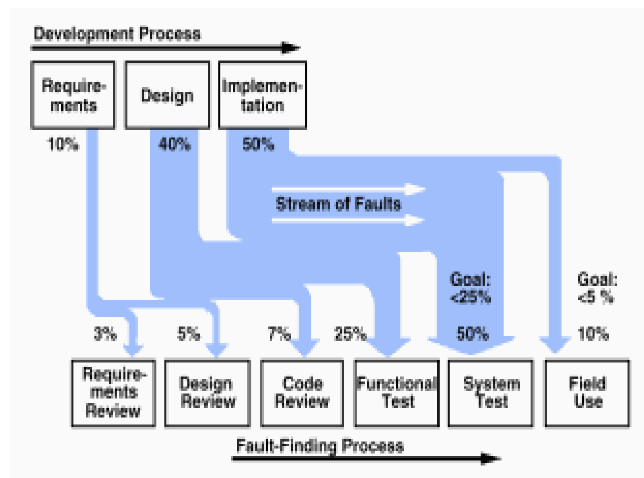

Fig. 7 Fault Injection and Fixing Model

The same NIST study also estimated that software defects cost the U.S. economy, alone, $60 billion per year. Many organizations view the software development lifecycle, in a Conventional way, as a linear process with discrete functions: design, develop, test and deploy. In reality, the software development lifecycle is a cyclical function with interdependent phases. Quality assurance has a role in every phase of that lifecycle, from requirements review and test planning, to code development and functional testing, to performance testing and on into production. It was unanimously agreed that quality and quality assurance is more than strictly testing at the end of the development process. Starting quality initiatives early and paying attention to quality throughout the development, deployment and production effort is key in order to achieve a baseline goal of zero-defect software.

5.4 Software Testing Economics Formulas

5.4.1 Techniques To Analyze Return On The Testing Investment (ROTI)

The ROTI model compares the development cost for a conventional project with the development cost for a project that uses TDD. The investment cost is the additional effort necessary to complete the TDD project as compared to the conventional project. The life cycle benefit is captured by the difference in quality measured by the number of defects that the TDD team finds and

fixes, but the conventional project does not. This defect difference is transformed into a monetary value using the additional developer effort corresponding to finding and fixing these defects in the conventional project.

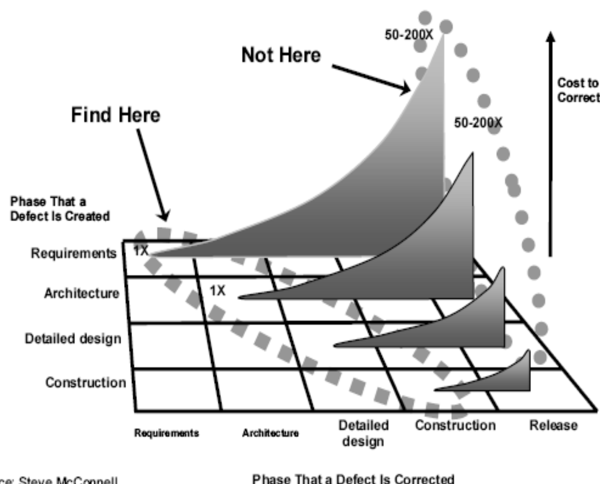

Fig. 8 Engineering Rules for Cost Of Defect Removal [10]

The concepts of the life cycle benefit and the investment cost in our context are depicted in Fig. 9. The upper horizontal line corresponds to the conventional project with additional quality assurance phase! The lower horizontal line corresponds to the TDD project. Our model captures the return on investment for an experienced TDD team in software testing process improvement (SPI).

5.4.2 Financial ROI

From a developer's perspective, there are two types of benefits that can accrue from the implementation of good software quality practices and tools: money and time. A financial ROI looks at cost savings and the schedule ROI than looks at schedule savings. Direct financial ROI is expressed in terms of effort since this is the largest cost on a software project. There are a number of different models that can be used to evaluate financial ROI for software quality. The first is the most common ROI model. We will show that this model is not appropriate because it does not accurately account for the benefits of investments in software projects. This does not mean that that model is not useful (for instance, accountants that we speak with do prefer the traditional model of ROI), only that we will not emphasize it in our calculations.

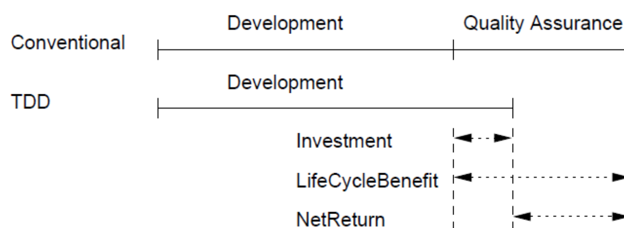

Fig. 9 Overview of benefit cost ratio calculation

Methods for return on investment (ROI) include benefit, cost, benefit/cost ratio, ROI, net present value, and breakeven point are given in Fig. 10.
ROI methods in general are quite easy, indispensable, powerfully simplistic, and absolutely necessary in the field of software process improvement (SPI). It is ironic that ROI methods are not in common practice. The literature does not abound with ROI methods for SPI. The ROI literature that does exist is very hard to locate, appears infrequently, and is often confusing. We also look at ROI at the project level, specially on return on the testing investment (ROTI), rather than at the enterprise level. ROI at the enterprise level (or across multiple projects) requires a slightly different approach which we will not address directly here. The most common ROI model, and that has been used more often than not in software engineering, is shown in the next formula:

$$ROTI_1 = \frac{Total \cdot CoQ \cdot Saved - Test \cdot Investment}{Test \cdot Investment} \quad (4)$$

Metric	Definition	Formula
Costs	Total amount of money spent on a new and improved software process	$\sum_{i=1}^{n} Cost_i$
Benefits	Total amount of money gained from a new and improved software process	$\sum_{i=1}^{n} Benefit_i$
B/CR	Ratio of benefits to costs	$\frac{Benefits}{Costs}$
ROI	Ratio of adjusted benefits to costs	$\frac{Benefits - Costs}{Costs} \times 100\%$
NPV	Discounted cash flows	$\sum_{i=1}^{Years} \frac{Benefits_i}{(1+Discount\ Rate)^{Years}} - Costs_0$
BEP	Point when benefits meet or exceed cost	$\frac{Costs}{Old\ Costs / New\ Costs - 1}$

Fig. 10 ROI metrics showing simplicity of ROI formulas and their order of application

This ROTI model gives how much the Total Cost of Quality (CoQ) savings gained from the project were compared to the initial investment. Let us look at a

couple of examples to show how this model works.

5.4.3 Schedule Benefits

If software quality actions are taken to reduce development cost, then this will also lead to a reduction in development schedule. We can easily calculate the reductions in the development schedule as a consequence of reductions in overall effort. In this section we will outline the schedule benefits of quality improvements. To do so we will use the schedule estimation model from COCOMO [11].

It is instructive to understand the relationship between project size and schedule as expressed in the COCOMO II model [11]. This is illustrated in Fig. 12. Here we see economies of scale for project schedule. This means that as the project size increases, the schedule does not increase as fast. The three lines indicate the schedule for projects employing different levels of practices. The lower risk and good practice projects tend to have a lower schedule.
Another way to formulate the ROTI model in Eqn. 4 which will prove to be handy is:

$$ROTI_2 = \frac{Original \cdot Total \cdot CoQ - New \cdot Total \cdot CoQ}{Original \cdot Total \cdot CoQ} \quad (5)$$

The *New Total CoQ* is defined as the total cost of software quality the project delivered after implementing the quality improvement practices or tools as in our work [9]. This includes the cost of the investment itself.
We can then formulate the *New Total CoQ* as follows:

$$New \cdot Total \cdot CoQ = Original \cdot Total \cdot CoQ \cdot (1 - ROTI_2)$$

Now, we can formulate the schedule reduction (ΔSCED or SCEDRED) as a fraction (or percentage) of the original schedule as follows:

$$\Delta SCED = \frac{Original \cdot Schedule - New \cdot Schedule}{Original \cdot Schedule} \quad (6)$$

By substituting the COCOMO equation for schedule, we now have:

$$\Delta SCED = \frac{PM_{Original}^{0.28+(0.002 \cdot \times \cdot \sum_{j=1}^{5} SF_j)} - PM_{New}^{0.28+(0.002 \cdot \times \cdot \sum_{j=1}^{5} SF_j)}}{PM_{Original}^{0.28+(0.002 \cdot \times \cdot \sum_{j=1}^{5} SF_j)}} \quad (7)$$

where:

$PM_{Original}$ - The original effort for the project in person-months

PM_{New} - The new effort for the project (after implementing quality practices) in person-months

SF_j - A series of five Scale Factors that are used to adjust the schedule (precedentedness, development flexibility, architecture / risk resolution, team cohesion, and process maturity).

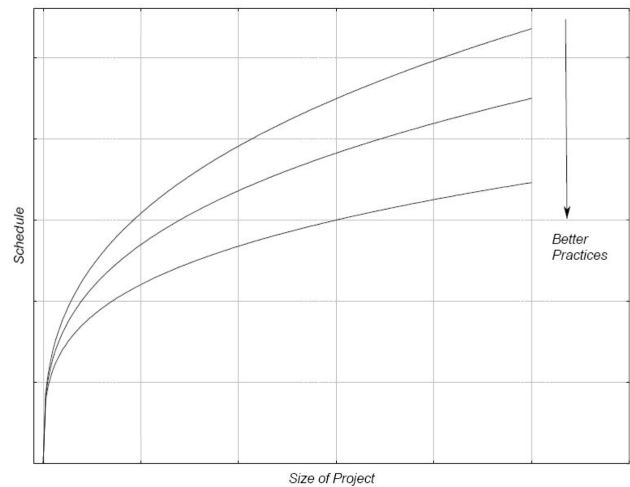

Fig. 12 Relationship between project size and schedule in COCOMO II.

Now, by making appropriate substitutions, we have:

$$\Delta SCED = \frac{PM_{Original}^{0.28+(0.002\times\sum_{j=1}^{5} SF_j)} - \left[PM_{Original}^{0.28+(0.002\times\sum_{j=1}^{5} SF_j)} \times (1 - ROTI_2)^{0.28+(0.002\times\sum_{j=1}^{5} SF_j)} \right]}{PM_{Original}^{0.28+(0.002\times\sum_{j=1}^{5} SF_j)}}$$

Which simplifies to:

$$\Delta SCED = 1 - (1 - ROTI_2)^{0.28 + (0.002 \times \sum_{j=1}^{5} SF_j)} \quad (8)$$

The relationship between cost savings and schedule reduction is shown in Fig. 13. As can be seen, the schedule benefits tend to be at smaller proportions than the cost benefits. Nevertheless, shaving off 10% or even 5% of your schedule can have nontrivial consequences on customer relationships and market positioning.

5.4.4 Interpreting The ROI Values

In this section we will explain how to interpret and use the ROI values that are calculated. First, it must be recognized that the ROI calculations, cost savings, and project costs as presented in our models are estimates. Inevitably, there is some uncertainty in these estimates. The uncertainty stems from the variables that are not accounted for in the models (there are many other factors that influence project costs, but it is not possible to account for all of these since the model would then be unusable). Another source of uncertainty is the input values themselves.

These values are typically averages calculated from historical data; to the extent that the future differs from the past these values will have some error. Second, note that the calculated ROI values are for a single project. A software organization will have multiple on-going and new projects. The total benefit of implementing software quality practices to the organization can be calculated by generalizing the results to the organization. For example, if the ROI for a single project is say a 15% saving. Assuming that the input values are the same for other projects in the organization, then we can generalize to the whole organization and estimate that if software quality practices are implemented on all projects in the organization, the overall savings would be 15%.

If the software budget for all the projects is say 20 million, then that would translate into an estimated saving of 3 million. Note that this is not an annual saving, but a saving in total project budgets hat may span multiple years (i.e., for the duration of the projects). To annualize it then the 15% savings must be allocated across multiple years. If you are implementing quality improvement on a single project, then these costs would have to be deducted from the single project savings. If you are implementing quality practices in the whole organization, then these costs will be spread across multiple projects.

6 Advanced Quantitative Defect Management (AQDM) Model

The answer to **RQ5** can be found in our work [7]. In this section we explain how can Quantitative Defect Management (QDM) Model be enhanced (as answer to **RQ4**) to be practically useful for determining which activities need to be addressed to improve the degree of early and cost-effective software fault detection with assured confidence, than definitely, **optimality** and **stability** criteria of very complex STP dynamics problem control (described in Section 7).

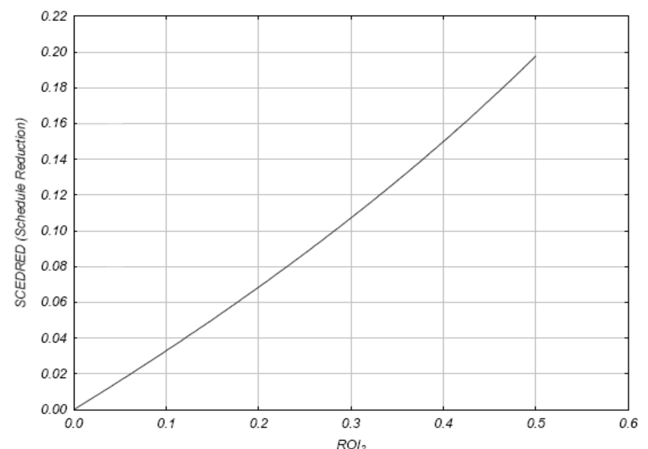

Fig. 13 The relationship between cost savings and schedule reduction for up to 50% cost savings. The assumption made for plotting this graph was that all Scale Factors were at their nominal values.

6.1 The defect containment measure

An error in an activity of development phase P_i (i=1 to N) is made that causes a failure (see Fig. 15-17). The failure leads to a reported anomaly. When the reported anomaly is analyzed, the fault(s) causing the failure is found and corrected. *Rework* is about revising an existing piece of software or related artifact. Therefore, a typical rework activity is to correct reported anomalies. Rework can be divided into two primary types of corrective work [9]:

• *Avoidable rework* is work that would not have been needed if the previous work would have been correct, complete, and consistent. Such rework consists of the effort spent on detecting and fixing software difficulties that could have been discovered earlier or avoided altogether [2,5].

• *Unavoidable rework* is work that could not have been avoided because the developers were not aware of or could not foresee the change when developing the software, e.g. changed user requirements or environmental constraints [9].

Using raw defect containment data and deriving AQDM measures early in the development life cycle provides opportunities for a project to identify issues in defect capture before costs spiral out of control and schedule delays ensue.

This section describes the selected method for how to achieve the objectives stated in the previous section. The method can be divided into the following three steps:

1. Determine which faults that should have been avoided or at least found earlier,
2. Determine the average cost of finding faults in different phases,
3. Determine the improvement potential from the results in (1) and (2).

The three sub-sections below describe how to perform each of the three steps.

6.2 The raw defect containment data

This section is dedicated to a model for assessing a plan for SQA defect-removal effectiveness and cost. The model, a multiple filtering model as shown on Fig. 7, is based on data acquired from a survey of defect origins, percentages of defect removal achieved by various quality assurance activities, and the defect-removal costs incurred at the various development phases. The model enables quantitative comparison of quality assurance policies as realized in quality assurance plans. The application of the proposed model is based on three types of data, described under the following headings from [1].

6.2.1 Defect removal effectiveness

It is assumed that any quality assurance activity filters (screens) a certain percentage of existing defects. It should be noted that in most cases, the percentage of removed defects is somewhat lower than the percentage of detected defects as some corrections (about 10% according to [4]) are ineffective or inadequate. The remaining defects, those undetected and uncorrected, are passed to successive development phases. The next quality assurance activity applied confronts a combination of defects: those remaining after previous quality assurance activities together with "new" defects, created in the current development phase. The main objective of the case study presented in this section was to investigate how fault statistics could be used for removing unnecessary rework in the software development process. This was achieved through a measure called Faults-Slip-Through (FST) [5,9], i.e. the measure tells which faults that would have been more cost-effective to find in earlier phases.

As previously mentioned, FST measurement was used for determining this, i.e. it evaluates whether each fault slipped through the phase where it should have been found or not. The main difference between FST measurement and other related measurements is when a fault is introduced in a certain phase but it is not efficient to find in the same phase. For example, a certain test technique might be required to simulate the behaviour of the function. Then it is not a fault slippage. Figure 14 further illustrates this difference. A consequence of how FST is measured is that a definition must be created to support the measurement, i.e. a definition that specifies which faults that should be found in which phase. To be able to specify this, the organization must first determine what should be tested in which phase. Therefore, this can be seen as test strategy work. Thus, experienced developers, testers and managers should be involved in the creation of the definition. The results of the case study in Section 6.2.2 further exemplify how to create such a definition.

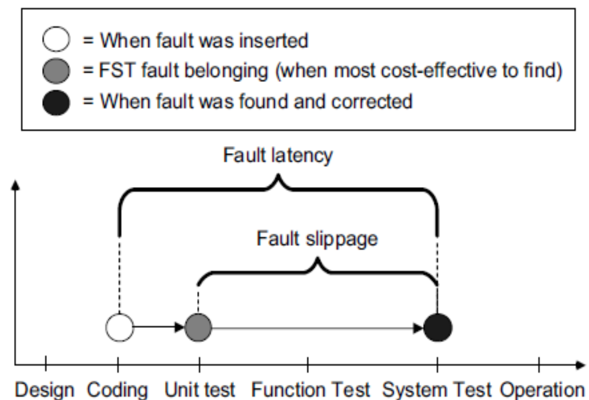

Fig. 14 Example of Fault Latency and FST

When having all the faults categorized, the next step is to estimate the cost of finding faults in different phases. From the measure, the improvement potential of different parts of the development process is estimated by calculating the cost of the faults that slipped through the phase where they should have been found (see Fig. 7 and 8 in our work[7]). The usefulness of the method was demonstrated by applying it on two completed development projects [1] and [2]. The results determined that the implementation phase had the largest improvement potential since it caused the largest FST cost to later phases, i.e. from 56 to 87 percent of the total improvement potential in the two studied project scenarios. It is assumed that the filtering effectiveness of accumulated defects of each quality assurance activity is not less than 40% (i.e., an activity removes at least 40% of the incoming defects). Typical average defect filtering effectiveness rates for the various quality assurance activities, by development phase, based on Boehm [11] and Jones [4], are listed in Table 2.

6.2.2 Cost of defect removal

Data collected about development project costs show that the cost of removal of detected defects

varies by development phase, while costs rise substantially as the development process proceeds. For example, removal of a design defect detected in the design phase may require an investment of 2.5 working days; removal of the same defect may require 40 working days during the acceptance tests. Several surveys carried out by IBM, TRW, GTE, Boehm and others, summarized by Boehm [11], estimate the relative costs of correcting errors at each development phase. Estimates of effectiveness of software quality assurance tools and relative costs of defect removal are provided by McConnell [10]. Although defect removal data are quite rare, professionals agree that the proportional costs of defect removal have remained constant since the surveys conducted in the 1970s and 1980s. Instead of average per phase defect removal cost we propose average relative defect-removal costs injected in phase $P_{i\ (i=1\ to\ 7)}$ and detected and removed latter in downstream phases P_j , $j>i$ up to the operation phase ($j=7$) as shown in Table 3.

6.2.3 Qunatitative Defect Removal Model

The model is based on the following assumptions:

■ The development process is linear and sequential, following the waterfall model of CMM Level 5. Software size is aproximately 100FP (1 injected defect/FP) i.e. for Java implementation about 50KLOC of source code [4].

■ A number of "new" defects are introduced in each development phase. For their distributions, see Fig. 15 and 16.

■ Review and test software quality assurance activities serve as filters, removing a percentage of the entering defects and letting the rest pass tothe next development phase. For example, if the number of incoming defects is 30, and the filtering efficiency is 60%, then 18 defects will beremoved, while 12 defects will remain and pass to be detected by the next quality assurance activity. Typical filtering effectiveness rates for the Standard quality assurance activities are shown in Table 2.

■ At each phase, the incoming defects are the sum of defects not removed by the former quality assurance activity together with the "new" defects introduced (created) in the current development phase.

■ The cost of defect removal is calculated for each quality assurance activity by multiplying the number of defects removed by the relative cost of removing a defect (see Table 3, 3rd column).

■ The remaining defects, unfortunately passed to the customer, will be detected by him or her. In these circumstances, full removal entails the heaviest of defect-removal costs.

Table 2 Average filtering (defect removal) effectiveness by Standard quality assurance activities plan [1]

No.	Quality assurance activity	Defect removal effectiveness	Cost of removing a detected defect (cost units)
1	Requirement specification review	50%	1
2	Design review	50%	2.5
3	Unit test – code	50%	6.5
4	Integration test	50%	16
5	Documentation review	50%	16
6	System test	50%	40
7	Operation phase	100%	110

In this model, each of the quality assurance activities is represented by a filter unit, as shown for Design in Fig. 15.

Table 3 Representative average relative defect-removal costs and fixing multiplier because FST

No.	Quality assurance activity	Average Cost of removing defects [cost units]	Fixing multiplier (Cost ratio) CM $_{P1\rightarrow P7}$	Fixing multiplier (Cost ratio) CM $_{P2\rightarrow P7}$	Fixing multiplier (Cost ratio) CM $_{P3\rightarrow P7}$	Fixing multiplier (Cost ratio) CM $_{P4\rightarrow P7}$	Fixing multiplier (Cost ratio) CM $_{P5\rightarrow P7}$	Fixing multiplier (Cost ratio) CM $_{P6\rightarrow P7}$
1	Requirement specification rev	1	1					
2	Design review	2.5	5	1				
3	Unit test – code	6.5	10	2	1			
4	Integration test	16	50	10	5	1		
5	Documentation review	16	130	26	13	3	1	
6	System test	40	368	64	37	7	3	1
7	Operation phase	110	400	75	40	20	15	10

The model presents the following quantities:

■ POD = Phase Originated Defects (from Fig. 16)

■ PD = Passed Defects (from former phase or former quality assurance activity)

■ %FE = % of Filtering Effectiveness (also termed % screening effectiveness) (from Table 2)

■ RD = Removed Defects

■ CDR = Average Cost of Defect Removal (from Table 2)

■ TRC = Total Removal Cost: TRC = RD ×CDR.

The illustration in Fig. 16 of the model applies to a standard quality assurance plan ("standard defects filtering system") that is composed of six

quality assurance activities (six filters), as shown in Table 2. A comprehensive quality assurance plan ("comprehensive defects filtering system") achieves the following: (1) Adds two quality assurance activities, so that the two are performed in the design phase as well as in the coding phase.

(2) Improves the "filtering" effectiveness of other quality assurance activities.

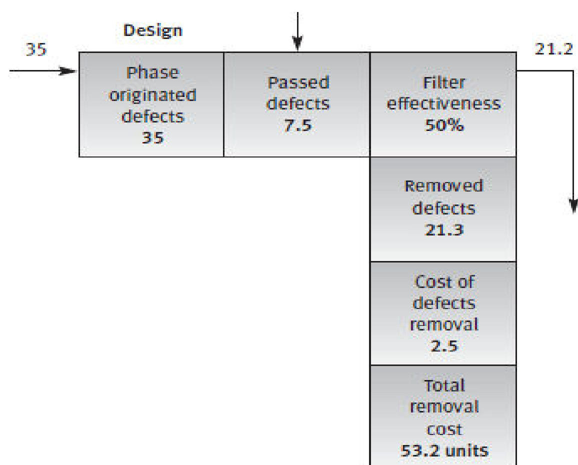

Fig. 15 A filter unit for defect-removal effectiveness: example (100 defects) from [1]

The comprehensive quality assurance plan can be characterized as shown in Table 4.

The main conclusions drawn from the comparison are:

(1) The standard plan successfully removes only 57.6% (28.8 defects out of 50) of the defects originated in the requirements and design phase, compared to 92.0% (46 defects out of 50) for the comprehensive plan, before coding begins.

(2) The comprehensive plan, as a whole, is much more economical than the standard plan as it saves 41% of total resources invested in defect removal, compared to the standard plan.

(3) Compared to the standard plan, the comprehensive plan makes a greater contribution to customer satisfaction by drastically reducing the rate of defects detected during regular operations (from 6.9 % to 3 %).

The comparison also supports the belief that additional investments in quality assurance activities yield substantial savings in defect removal costs. Alternative models dealing with the cumulative effects of several qualityassurance activities are discussed by [2,5,9] as described below. A process-oriented illustration of the comprehensive quality assurance plan and model of the process of removing 100 defects is provided in Fig. 17. A comparison of the outcomes of the standard software quality plan versus the comprehensive plan is revealing as shown in Table 5.

Table 4 Comprehensive quality assurance plan [1]

No.	Quality assurance activity	Defect-removal effectiveness	Cost of removing a detected defect (cost units)
1	Requirement specification review	60%	1
2	Design inspection	70%	2.5
3	Design review	60%	2.5
4	Code inspection	70%	6.5
5	Unit test – code	40%	6.5
6	Integration test	60%	16
7	Documentation review	60%	16
8	System test	60%	40
9	Operation phase	100%	110

In general, the quantitative results of the comparison comply nicely with the SQA approach.

Table 5 Comparison of the standard and comprehensive quality assurance plans

No.	Quality assurance activity	Standard plan		Comprehensive plan	
		Percentage of removed defects	Cost of removing defects (cost units)	Percentage of removed defects	Cost of removing defects (cost units)
1	Requirements specification review	7.5%	7.5	9%	9
2	Design inspection	–	–	28.7%	71.8
3	Design review	21.3%	53.2	7.4%	18.5
4	Code inspection	–	–	24.4%	158.6
5	Unit test – code	25.6%	166.4	4.2%	27.3
6	Integration test	17.8%	284.8	9.8%	156.8
7	Documentation review	13.9%	222.4	9.9%	158.4
8	System test	7.0%	280	4%	160
	Total for internal quality assurance activities	**93.1%**	**1014.3**	**97.4%**	**760.4**
	Defects detected during operation	6.9%	759	2.6%	286
	Total	**100.0%**	**1773.3**	**100.0%**	**1046.4**

6.3 Simulation results of AQDM improvement

Unlike conventional approaches to software testing which are applied to the software under test without an explicit optimization goal, as described above, the *OptimalSQM* approach designs an optimal testing strategy to achieve an explicit optimization goal, given a priori is described in our works [5,6].

We described in this section, as answer to the **RQ4**, a Software Quality Optimization (SQO) strategy of *OptimalSQM* framework, which is a continuous, iterative process throughout the application lifecycle resulting in zero-defect software that delivers value from the moment it goes

live, with Simulated Defect Removal Cost Savings model using net savings that are calculated using this formulae:

$$NS = FST_{r \to P+1} * (CM_{r \to P+1} - CM_{r \to P}),\ r=1..6$$

for the given large (~11300 FP, Java implementation about 600KLOC of source code) project example from [2]. The results determined that the implementation phase (P3) had the largest improvement potential since it caused the largest FST cost to later phases, i.e. from 56 to 87 percent of the total improvement potential in the two studied project scenarios.

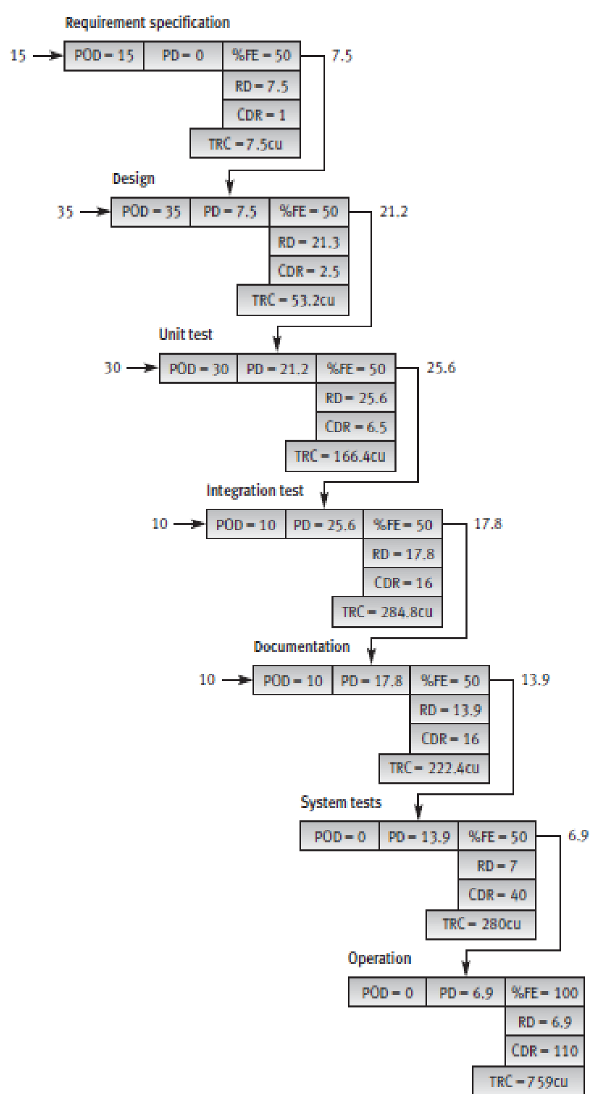

Fig. 16 DRE and costs of Standard QA plan and model of the process of removing 100 defects [1]

7 Optimality and stability criteria of STP dynamics problem control

A basic rule from cybernetics - that a long time lag between the output signal from the controlled system and feedback to the controller causes instability in the system - applies to SDP-STP processes as well. Long design iteration loops with late feedback drive cost and schedule overruns in SDP-STP.

In order that *OptimalSQM* framework to be practically useful for determining which activities need to be addressed to improve the degree of early and cost-effective software fault detection with assured confidence, than definitely, **optimality** and **stability** criteria of very complex STP dynamics problem control is described in this Section as answer to **RQ5.** How should a software development organization apply the metric(s) suggested above for assessing ongoing and finished projects we propose one **Dynamic Control Model** of SDP-STP in Fig. 18 with **optimality** and **stability** criteria of very complex STP dynamics problem control described in next sub-sections.

DEFECTS →			FOIND IN PHASE :							
			P_1	P_2	P_3	P_4	P_5	P_6	P_7	
			Requirement	HL Design	LL Design	Code (Unit test)	Integration/ System test	Acceptance (User test)	Operation Post - release	Total Injected in phase
PHASE INSERTED :	P_1	Requirement	9	6	0	0	0	0	0	15
	P_2	HL Design	0	22	13	0	0	0	0	35
	P_3	LL Design	0	0	7	4	0	0	0	11
	P_4	Code (Unit test)	0	0	0	13	6	0	0	19
	P_5	Integration/ System test	0	0	0	0	3	7	0	10
	P_6	Acceptance (User test)	0	0	0	0	0	7	0	7
	P_7	Operation Post - release	0	0	0	0	0	0	3	3
		Number of Defects found and removed in phase	9	28	20	17	9	14	3	**100**
		Defect Removal Efficiency [%] In phase	60.0	56.0	32.8	21.3	10.0	14.4	3.0	**Total defects**
		Cumulative Removal Efficiency [%]	*9.0*	*37.0*	*57.0*	*74.0*	*83.0*	***97.0***	Total DRE [%]	

Fig. 17 DRE of Comprehensive QA plan and model of removing 100 defects [1]

When design and testing activities are not coupled, the information testing provides on product design is delivered at a wrong point in the process. This late information is either not useful any more or shows design problems too late, causing undesired late rework. Thus, iteration cycles should be kept short and rapid. However, this is difficult in the context of a number of interrelated activities without a model to facilitate process analysis and improvement.

Planning, managing, executing, and documenting testing as a key process activity during

all stages of development is an incredibly difficult process.

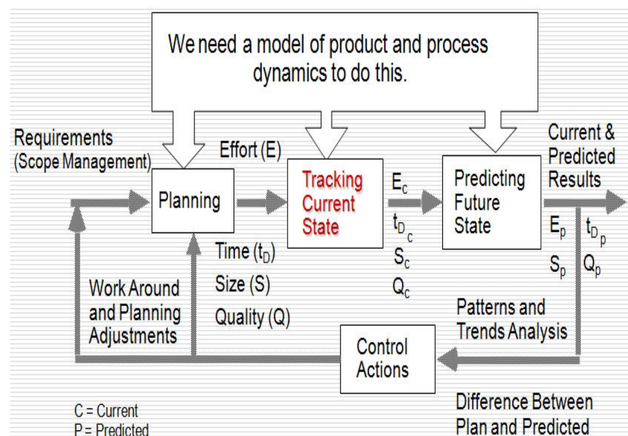

Fig. 18 The feedback control model for SDP-STP

There is strong demand for software testing effectiveness and efficiency increases with Planning, Estimating SDP-STP [1-9,18,19] and Predicting (PEP): Time schedule, Size and tracking current project software Quality metrics with explored the true costs of software defects and their impact on application performance; demonstrated *OptimalSQM* framework simulation to find how quality processes implemented throughout the application lifecycle can result in measurable performance improvements; presented economic model for the return on investment of TDD (ROTI) based on a variety of ways of calculating ROI (described in previous Sections).

This section describes the selected method for how to achieve the objectives stated in the previous section. The method can be divided into the following three steps:

1. Determine the Life Cycle Benefit i.e. Optimality criteria,
2. Determine the statistical control limits for estimated *OptimalSQM* metrics,
3. Determine the execution confidence from the results in (1) and (2).

The three sub-sections below describe how to perform each of the three steps.

7.1 The Life Cycle Benefit model parameters formulas for calculations

This section describes those formulas of our *OptimalSQM* metrics model which are necessary to understand the break-even and ROTI analysis if the investment of described STP improvements in previous sections pays off.

Calculating the return on investment ROI means to add up all the benefits of the investment, subtract the cost, and then compute the ratio of the cost according the equation (6) in Section 5.4.2 Financial ROI). If the investment in STP improvement pays off, the $ROTI_1$ is positive, otherwise negative. In our evaluation of TDD we focus on the benefit cost ratio BCR which is easily derived from the return on investment.

$$BCR = LifeCycleBenefit/Investment = ROTI_1 + 1$$

Studying the BCR instead of the $ROTI_1$ makes the break-even analysis much simpler, see below.

7.1.1 Investment Cost

We first look at the investment cost. For the conventional project, the development phase includes design, implementation and test. The development phase of the TDD project is comprised only of test-driven development.

As first empirical evidence suggests, we assume that the TDD project lasts longer than the conventional project. We call the ratio of the project durations the test-speed-disadvantage (TSD).

$$TSD = Time_{Conv}/Time_{TDD}$$

Since we assume that the development phase is shorter for the conventional project, because include small number of test activities, the test-speed-disadvantage ranges between 0 and 1:0 < TSD < 1.

Using productivity figures to explain the difference in elapsed development time between the two kinds of project, the TDD development is (1 − TSD) × 100 % less productive than the conventional project. Finally, the investment is the difference between the development cost of the TDD project and the conventional project as depicted in Fig. 9.

7.1.2 Life Cycle Benefit

Now, we consider the benefit. Each development process is characterized by a distinct defect-removal-efficiency -DRE (recall the section 5.2). The defect-removal-efficiency denotes the percentage of defects a developer eliminates during development. Initially, a developer inserts a fixed amount of defects per thousands lines of code (initial-defect-density, IDD), but he eliminates DRE × 100 % of the defects during the

development process. From the increased reliability assumed for TDD,we have:

$0 < \text{DRE}_{\text{Conv}} < \text{DRE}_{\text{TDD}} < 1.$

The additional quality assurance (QA) phase of the conventional project compensates for the reduced defect-removal-efficiency of the conventional process. The only purpose of the Comprehensive QA plan phase is to remove all those defects found by TDD but not by the conventional process (recall the section 6.2). The amount of defects to be removed in the Comprehensive QA plan phase is mainly characterized by:

$\Delta\text{DRE} = \text{DRE}_{\text{TDD}} - \text{DRE}_{\text{Conv}}$.

The benefit of TDD is equal to the cost of the Comprehensive QA plan phase for the conventional project. The benefit depends on the effort (measured in developer months) for repairing one line of code during QA, which is characterized by

$$\text{QA}_{\text{Effort}} = \frac{DRT * IDD}{WT}$$

$\text{QA}_{\text{Effort}}$ depends on the following:

- The defect removal time DRT. It describes the developer effort in hours for detecting (finding) and removing one defect.
- The inital defect density IDD. The number of defects per line of code inserted during development.
- The working time WT. The working hours per month of a developer. The reciprocal of $\text{QA}_{\text{Effort}}$ is a measure for the productivity during the QA phase.

7.1.3 Benefit Cost Ratio

The benefit cost ratio is the ratio of the benefit and the investment. Substituting the detailed formulas given in [24] of our model, the benefit cost ratio becomes:

$$\text{BCR} = \frac{QA_{Effort} * \text{Prod} * \Delta DRE * TSD}{(1 - TSD)} \qquad (9)$$

Where, **Prod** is the productivity of the conventional project during the development phase measured in lines of code per month. Values larger than 1 for the BCR mean a monetary gain from TDD, values smaller than 1 a loss.

7.1.4 Break Even

Setting the benefit cost ratio equal to 1, we get a relation between the test-speed-disadvantage of TDD and the reliability gain of TDD:

$$\text{TSD} = \frac{1}{c * \Delta DRE + 1}, \text{ or}$$

$$\Delta DRE = \frac{1 - TSD}{c * TSD}, \text{ where } c = QA_{Effort} * \text{Prod}$$

As an example, we examine the benefit cost ratio of the following scenario.

Factor	Value
DRT	10 h/defect
IDD	0.1 defects/LOC
WT	135 h/month
Prod	350 LOC/month

Let TSD and ΔDRE vary. Figure 19 shows the benefit cost ratio plane spanned by the test-speed-disadvantage TSD and the defect-removal-efficiency difference ΔDRE. Values larger than 4 are cut off.

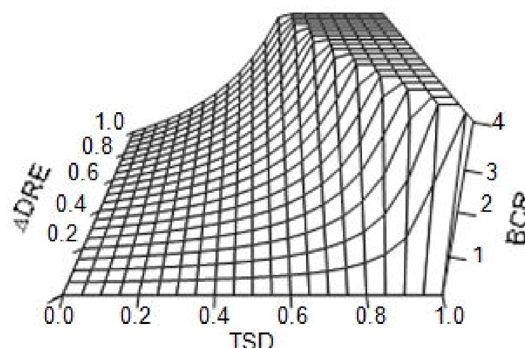

Fig. 19 Benefit cost ratio dependent on TSD and ΔDRE

For large values of the test-speed-disadvantage (TSD > 0.9) the TDD project performs almost always better than the conventional project, even for a small defect-removal-efficiency difference. On the other hand, if the test-speed-disadvantage is very small (TSD < 0.2), TDD does not produce any benefit regardless how large the defect-removal-efficiency difference is.

The TSD can be estimated with formula (8) for ΔSCED in Section 5.4.3. The relationship between cost savings defined by $ROTI_2$ and schedule reduction is shown in Fig. 13 in the same Section.

Next section describes the selected method for how to achieve the objectives stated in the previous section i.e. step two.

7.2 Statistical control limits for estimated *OptimalSQM* metrics

Advanced Quantitative process management implemented in *OptimalSQM* framework is among the advanced features of highly mature processes as defined in capability maturity model integration

(CMMI), which provides insights on the degree of goal fulfillment and root causes of significant process/product deviation [25]. Quantitative defects management predicts the number of defects expected to be detected in each stage of software development, enabling proactive measures to be taken early in development [26]. Quantitative defects management is the key to ensure the production of high-quality software, which has been an important part of quantitative process management. Unfortunately, how to quantitatively manage defects across multiple test iterations remains a challenging issue [26]. *Two process areas (OPP, organizational process performance; QPM, quantitative project management) and some statistical techniques (e.g. statistical process control) are described in CMMI for implementing quantitative process management.* However, most software organizations still do not know clearly how to apply quantitative process management. Therefore, detailed, experience-based guidance would be helpful for software organizations applying or planning to apply quantitative process management.

In this section, we introduce a process performance Baselines (PPBs) based on Advanced Quantitative Defects Management (AQDM) method [2] and its application in a Chinese telecommunications company (named ZZNode) published in [26] which we adapted for our purpose in *OptimalSQM* framework. The AQDM method covers all defect detection activities, e.g. review, inspection, and testing which has successfully applied in *OptimalSQM* framework in quantitative defect control as depicted in Fig. 20.

AQDM method of quantitatively managing the testing process, which supports high-level process management mentioned in CMMI. As shown in Fig. 1, the four steps of the AQDM method are to: (1) identify the performance objectives (P-Objs) to be managed quantitatively and construct data samples; (2) establish the P-BL for the identified P-Objs; (3) establish the process-performance model for fixing effort; and (4) establish the process-performance model for fixing schedule.

As shown in Fig. 20, by using the methodology, the empirically based models for the testing process can be established based on the analysis of historical data, which will be described in next Section. Software projects can use the model to estimate and control the defects, effort and schedule quantitatively.

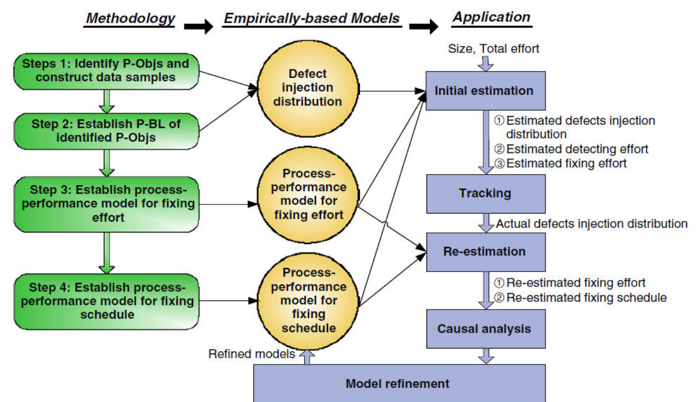

Fig. 20 The illustration of the AQM method [26]

7.2.1 Identify P-Objs and Construct Data Samples

Normally, the effort of detecting and fixing defects, and the defect-injected phase are sensitive data that we should consider for testing process. A general assumption is that the effort of detecting and fixing defects should consume a certain percentage in the total development effort, and the effort of fixing defects is influenced by the defect number and the defect-injected phase. In the AQDM method, four P-Objs have been identified as follows:

1. **Percentage of Detecting Effort ($\%Eff_{Detect}$) or PDE:** Detecting effort means the effort for all detecting activities including test planning, test case preparation, test implementation and fix verification. The percentage of the detecting effort in the total effort is $\%Eff_{Detect}$.

2. **Defect Injection Distribution (DID):** In general, many software organizations collect defect data for quality control. There are always some defects injected in the early phases, which are only detected during the testing activities, even in high-maturity organizations. In our method, three primary phases, namely requirements, design and coding, are used to classify the corresponding injected phases for each defect. The corresponding percentages of defects injected in these phases are denoted as: requirements, ($\%DI_{Req}$); design, ($\%DI_{Design}$); and coding, ($\%DI_{Code}$) respectively. The principles of assigning the injected phase are described as: (i) defect injected in the requirements phase: a defect that is due to poor requirements, such as inconsistent and unclear requirements;(ii) a defect injected in the design phase: a defect that is due to poor design, such as unclear interface, misunderstanding of requirements and incomplete data verification; and (iii) a defect injected in the coding phase: a defect that is due to poor coding, such as incorrect words in

a Web page and inconsistent code against requirements or design.

3. **Schedule Factor for Defect Fixing (SF_{Fix}).** For each defect, the opening date is the day the defect is being submitted, and the closing date is the day the defect is being confirmed as repaired. The schedule of defect fixing ($Sced_{Fix}$) can be calculated by the formula below.

$Sced_{Fix}$ = *closing date − opening date + 1.*

Sometimes, certain defects are assigned 'deferred' and not to be fixed in the current release due to business pressures. In this case, we take the day the defect is being deferred and calculate the $Sced_{Fix}$ as shown below.

$Sced_{Fix}$ = *deferred date − opening date + 1.*

The $Sced_{Fix}$ for deferred defects means the schedule of the defect being dealt with.
For each project, the average schedule of fixing one defect ($ASced_{Fix}$) injected in each phase can be calculated by the formula below.

$ASced_{Fix}$ *of each phase = total* $Sced_{Fix}$*/ total defects injected in the phase*

Normally, the $ASced_{Fix}$ of coding phase ($ASced_{Code}$) is the shortest. We use the $ASced_{Code}$ as the benchmark (i.e. SF_{Code}), and calculate the ratio of $ASced_{Fix}$ of requirements phase ($ASced_{Req}$) to $ASced_{Code}$, as well as the ratio of $ASced_{Fix}$ of design phase ($ASced_{Design}$) to $ASced_{Code}$ by the formula below borowed from [26].
$SF_{Code} = 1$
$SF_{Req} = ASced_{Req}/ASced_{Code}$
$SF_{Design} = ASced_{Design}/ASced_{Code}$
As we mentioned before, many software projects are delayed due to the slippage of the testing process. In fact, many testing processes are delayed due to the schedule overrun of defect-fixing activity. To solve this problem, AQDM uses a more effective method on estimating schedule of defect-fixing activity [5,26]. An algorithm to help estimate the schedule of defect-fixing activity is established based on the analysis of SF_{Fix} and the effort of defect fixing. The algorithm applies the following principles:
• *Shortest schedule.* Based on the total effort of defect fixing, the defect-fixing schedule should be as short as possible.
• *Concurrent defect fixing.* Defects which require a long fixing schedule should be fixed concurrently if there are sufficient human resources available.

The basic ideas of the algorithm are: (1) the fixing schedule of defects injected in requirements should be allocated first since the $ASced_{Req}$ is the longest, which is the basis of the fixing schedules of defects injected in design and coding; (2) if the number of defects injected in the design and the number of defects injected in the coding are similar, as well as if the SF_{Req} is longer than the sum of SF_{Design} and SF_{Code}, then the fixing schedule of defects injected in design and coding could be allocated serially. Especially, in the algorithm, we assume that if ½ < *numbers of defects injected in design/numbers of defects injected in coding* <2, it means that the numbers of defects injected in design and coding are similar; and (3) in the other cases, the schedule of defects injected in design and the schedule of defects injected in coding should be allocated concurrently. According to P-BLs of SF_{Fix} , published in [26] it is obvious that the earlier in the phase the defects get injected, the longer is the schedule needed to fix the defects.
Based on the $ASced_{Fix}$ of the 16 projects and the P-BL of SF^{Fix}, the organization defined some rules for defects management as shown in Table 6. Based on the P-BLs of SF_{Fix}, we establish the process-performance model for fixing the schedule.

4. **Percentage of Fixing Effort ($\%Eff_{Fix}$) or PFE:** Fixing effort data means the effort for all defect-fixing activities including defect analysis and fixing. $\%Eff_{Fix}$ is the percentage of the fixing effort in the total effort.

Defect fixing is an important activity of software development, which demands a certain amount of effort. In the International Software Benchmark Standard Group (ISBSG), (www.isbsg.org), the fixing effort is collected and counted in rework effort. However, many effort estimation methods do not pay sufficient attention to the effort of defect fixing; instead, they just include it in the testing activities. Normally, defect detecting is performed by a testing team, and defect fixing is performed by a development team. Estimating their effort separately is helpful for an organization to plan its human resources and schedules. In addition, the fixing effort is strongly correlated with the number and injected phase of defects. Splitting them and establishing their P-BLs are very useful to manage testing process quantitatively.

In CMMI, the process-performance model is a description of the relationships among attributes of a process and its work products that are developed from historical process-performance data, and calibrated using collected process and product measures from the project, and are used to predict results to be achieved by following a process [25].

Table 6 Defects management rules for fixing the schedule

Rules	Description
Early detection principle	Defects should be detected as early as possible.
Limitation of defect-fixing schedule	Defects injected in the requirement phase should be fixed in 8 days; the control limits are 5–11 days.
	Defects injected in the design phase should be fixed in 5 days; the control limits are 3–7 days.
	Defects injected in the coding phase should be fixed in 2 days; the control limits are 1–3 days.

In the testing activities, there is a consensus that the earlier a defect is injected, the more effort is needed to fix it as we described in previous sections. In contrast, the later a defect is injected, the less effort is needed to fix it. So, defects injected in an earlier phase, such as the requirements phase, have the effect of increasing the defect-fixing effort,whereas, defects injected in a later phase, such as the coding phase, have the effect of decreasing the defect-fixing effort (recall the Table 2 in 6.2.3 Qunatitative Defect Removal Model). After constructing defect-related data samples, software organizations can discover some more precise correlation between defects and fixing effort. The process-performance model for fixing effort is based on this hypothesis. There are some statistical methods which can be used to analyze the correlation between DID and $\%Eff_{Fix}$, such as multiple regression analysis. After the correlation between DID and $\%Eff_{Fix}$ has been analyzed, the regression equation between DID and $\%Eff_{Fix}$ can be used to refine the estimation of fixing effort after testing. The outcome can provide a guideline to estimate the effort of defect fixing based on the defects and the distribution of injection phases. So, after testing, project managers could reestimate and replan their fixing effort effectively. The factors of regression equation could be refined and calibrated based on the historical data of software organizations. Thereafter, it can be better applied in these organizations. We adopted factors of regression equation derived from the the historical data and for the fixing model as follows [26]:

$$\%Eff_{Fix} = 0.1065 \times \%DI_{Req} - 0.0043 \times \%DI_{Design} - 0.3925 \times \%DI_{Coe} + 0.3597$$

For high-maturity software organizations, the defect-related process performance, such as defect injection, defect removal, and defect density, also has some common and stable properties. Many methods discuss the defect removal ratio and defect density. These are very useful and easy to understand. Here we focus on the defect injection and the correlation between the defects and effort needed to fix them.

Process Performance Baselines - PPB is a measurement of performance for the organization's set of standard processes at various levels of detail, as appropriate [25]. When all functions are coded and passed unit testing, integration testing ($I_{nt}T$) for all functions is performed. After all iterations are finished, product integration and system testing ($I_{nt}\&S_{ys}T$) for products developed by all iterations are performed.

Before establishing PPBs, ten defect related measures that provide appropriate insight into the project's quality and process performance, as shown in Table 7 were selected. The principles of selecting these measures are:

1. the measures can be collected easily, e.g. there is tool support for data collection;

2. the measures are closely related to the Advanced Quantitative process management implemented in *OptimalSQM* framework's objectives of development projects.

In Table 7, the measure numbers 1–4 focus on all kinds of defects, including defects detected in review, inspection, unit testing, integration testing, system testing, etc.; the measure numbers 6, 9, and 10 just focus on defects detected in system-testing activity. In Table 7, the measure number 1 is used to manage the distribution of defect injection in different kinds of activities which is applied in the DRE model; the measure number 2 is used to management the effectiveness of defect removal activities which is applied in the DRE model; the measures numbers 3 and 4 are used to manage the quality of product; the measure number 5 describes software productivity of the project; the measure number 6–8 are used to management system-testing activity which are applied in the fixing model; the measures numbers 9 and 10 describe the efficiency of testing and rework activities which are applied in the DRE model. PPB contains two important indicators: process performance and capability. The process performance is a measure of actual results achieved by following a process, specified by central line (CL). The process capability is *the range of expected results* that can be achieved by following a process, specified by Upper Control Limit (UCL) and Lower Control Limit (LCL). We

use the baseline – statistic – refinement method [26] and the XmR (individuals and moving range) control chart to establish PPBs. Due to space limit, we do not describe the process of calculating CL, UCL, and LCL in detail.

Table 7 Measures that should be collected in iterative development projects

No.	Measures
1	Defect injection rate of requirements, design, coding, and testing activities = number of defects injected at the activity/total number of defects of the project
2	Defect removal effectiveness of requirements, design, coding, and testing activities = number of defects removed at the activity/(number of defects existing on activity entry + number of defects injected during development of the activity)
3	Pre-release defect density = number of defects removed before product release/product size
4	Post-release defect density = number of defects detected within 1 year after product release/product size
5	Productivity = product size/total effort of project
6	Defect injection distribution = number of defects injected in requirements (or design, coding, and testing)/total number of defects removed in system testing × 100%
7	Percentage of detecting effort = effort of defect-detecting activity in system-testing stage/total effort of project × 100%
8	Percentage of fixing effort = effort of defect-fixing activity in system-testing stage/total effort of project × 100%
9	Test efficiency = number of defects/defect-detecting effort
10	Rework efficiency = number of defects/defect-fixing effort

7.2.2 The Control Limits for of a Process - Performance Model for Fixing Effort

For the 16 projects in Web application domain, from the the historical data published in [26], all the defects considered were detected in the testing activities. These defects were classified into four categories: critical defects, serious defects, noncritical defects and cosmetic defects.
In this article, we only describe the total defects collected without distinguishing them. The XmR (individuals and moving range) control chart is applied to analyze the DID (Defect Injection Distribution) data. Assume that the sequence of data sample is Xi, the moving range (mR) is:

$$mR_i = |X_i - X_{i-1}| \text{ , } i = 2 \ldots n$$

According to the theory of statistics, we can get the upper control limit (UCL), central line (CL), and lower control limit (LCL) for mR-chart and X-chart as follows:

$$UCL_{mR} = 3.268\overline{mR} \cdot CL_{mR} = \overline{mR} \cdot LCL_{mR} = 0$$

$$UCL_x = \overline{X} + 2.660\overline{mR} \cdot CL_x = \overline{X} \cdot LCL_x = \overline{X} - 2.660\overline{mR}$$

Figures 21-23 show the XmR control charts for $\%DI_{Req}$, $\%DI_{Design}$ and $\%DI_{Code}$ respectively. For the three XmR charts in Figures 21–23, all data points are distributed between the UCL and the LCL in both mR-chart and X-chart. Hence, the $\%DI_{Req}$, $\%DI_{Design}$ and $\%DI_{Code}$ were converged and the distribution of defect injection appears to be stable.
In telecommunication application domain as a reference, for *the range of expected results,* you can use data from Table 8.
We applied this empirical method on an ongoing project of the organization to estimate, plan and manage its testing process quantitatively. The P-BLs and correlation established above plus (recall previous sections) some other baselines to compose the Advanced Quantitative process management implemented in *OptimalSQM* framework of the organization process management system.

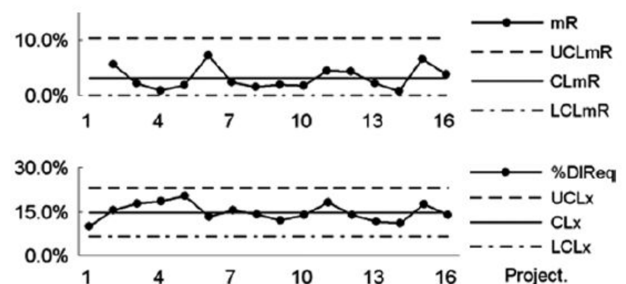

Fig. 21 XmR chart for $\%DI_{Req}$ data of the 16 projects [26]

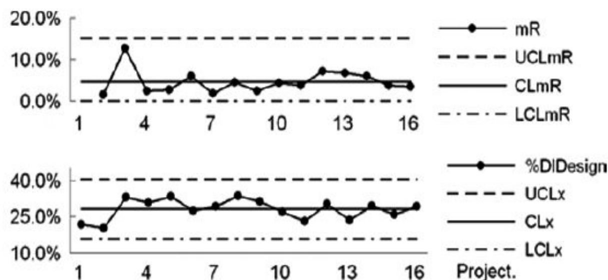

Fig. 22 XmR chart for $\%DI_{Design}$ data of the 16 projects [26]

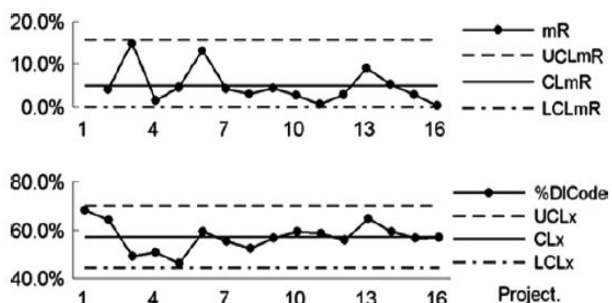

Fig. 23 XmR chart for $\%DI_{Code}$ data of the 16 projects [26]

The steps of applying the quantitative management model for testing process are: (1) based on the P-BL of the P-Objs, estimating the defect detecting effort, defect fixing effort and number of defects injected in each phases during the project planning; (2) through the testing activities, collecting the defect related data and re-estimating the effort of defect fixing when the actual P-Objs has abnormality which we can reveal by the Statistical-Risk-Based Test with Assured Confidence, explained in next sub-section.

7.3 The Statistical-Risk-Based Test with Assured Confidence as a stability criteria

In order to prevent endless regresion test of defect detection and fixing loop (recall the Fig. 17, The feedback control model for SDP-STP) i.e. abnormality of planned test activities and established control limits we must before the process starts, determine the threshold failure density and the corresponding confidence level. The threshold failure density must be determined by the requirements only. The number of daily test and fixing transactions and criticality of failures determine the threshold failure density. For example, if the system is mission-critical and no failure can be tolerated, the threshold should be low, say 0.0001.

Table 8 PPBs for telecommunication projects

PPBs		UCL	LCL	CL
DIR (%)	Requirements	64.1%	8.6%	36.4%
	Design	20.2%	8.0%	14.1%
	Coding	76.0%	13.6%	44.8%
	Testing	9.2%	0.2%	4.7%
DRE (%)	Requirements	85.9%	25.6%	55.8%
	Design	93.9%	2.4%	48.1%
	Coding	52.0%	0.0%	25.5%
	Testing	90.9%	28.0%	59.5%
Pre-release defect density (defects / KLOC)		4.07	3.40	3.70
Post-releases defect density (defects / KLOC)		1.02	0.99	1.01
Software productivity (LOC / Labor Day)		69.20	25.20	47.20
DID (%)	Requirements	9.7%	5.2%	7.4%
	Design	12.2%	5.6%	8.9%
	Coding	80.4%	61.0%	70.7%
	Testing	15.9%	10.1%	13.0%
PDE (%)		2.4%	0.8%	1.6%
PFE (%)		1.6%	0.4%	1.0%
Test efficiency (defects / labor day)		5.59	0.00	2.29
Rework efficiency (defects / labor day)		5.60	0.00	2.61

Once the threshold failure density is determined, the confidence level can be determined, and this again can be determined by the requirements. Note that higher confidence level and lower threshold on failure density increase the number of test cases needed. This process starts from module testing, to integration testing, and finally to end-to-end testing, and this process can be easily embedded in most software development processes. If a module fails at any stage of the **S**tatistical-**R**isk-**B**ased **T**est with **A**ssured **C**onfidence (SRBTAC) testing [8], it should be subjected to software modification and testing before it can be used for the next phase of the SRBTAC testing. Only when a module passes the SRBTAC module testing, it can be subjected to the SRBTAC integration testing. Similarly, only when a module pass the SRBTAC integration testing, it can be subjected to the SRBTAC end-to-end testing. This process has a feedback mechanism: if a fault is detected in integration, the corresponding module(s) must be subjected to another round of testing. The process helps in identifying areas that need further testing and/or rework.

7.3.1 SRBTAC Statistical Model (I)

The statistical model requires that test cases for the SRBTAC must be selected randomly and

independently. However, completely random test cases may not cover all the important partitions in the input domain. Thus, this paper recommends that the input domains are thoroughly analyzed to identify major partitions, and then test cases are generated from these partitions to ensure coverage. Major partitions can be identified by examining the constraints on inputs, outputs and major execution paths in the code or design and avoid any apparently dependent test cases. At each level of SRBTAC testing, only test cases from that level can be used. For example, at the SRBTAC end-to-end testing, only end-to-end test cases can be counted, but not integration or module test cases. In addition to regression testing, new test cases can be developed by composing and reusing existing test cases. During integration testing and end-to-end testing, the operating environment should be considered, and this may include external systems interfacing, physical environment, input data, system operators and end users. For each factor, identify those that are SRBTAC related and identify contingency plans for environment components that can not be certified.

The following equation is used to calculate the number of test case required to achieve a certain level of confidence ***C*** that the failure density is no more than a desired bound ***B***. All ***N*** test cases must execute correctly without causing the software to fail.

$$C = 1 - (1-B)^N \tag{10}$$

Where , **C** is the confidence level desired, **B** is the failure density(threshold), **N** is the number of test cases required and N = Ln(1-C)/Ln(1-B).
For example, if desired confidence level is *0.95*, and the target failure rate *0.05*, we need *58* test cases, because *0.95 = 1 - (1- 0.05)N* , so *N* needs to be *58*. The computation can be done using Microsoft's Excel or calculators to prepare table 1 that can be handy too.
In this process, at each level, we need 58 test cases to certify that the target software achieved 0.95 confidence with 0.05 failure density.
The formula (10) is applicable when every test case is successful. If one or more test cases fail, Statistical Model (II) shoul be used.

7.3.2 SRBTAC Statistical Model (II)

During the SRBTAC experiments at several testing sites, it is apparent that some of the testing projects have some failures, but it is still necessary to compute the confidence.

Table 9 The number of test cases required for various *C* and *B* when there is no failure.

Confidence Level(C)	Failure Density(B)	Number of test cases required
0.8	0.01	160
0.8	0.02	80
0.8	0.05	31
0.8	0.10	19
0.95	0.01	298
0.95	0.02	148
0.95	0.05	58
0.95	0.10	28

Thus, the formula is now changed to:

$$C = 1 - F = 1 - \sum_{k=0}^{Q} \binom{N}{k} B^k (1-B)^{N-k} = \sum_{k=Q+1}^{N} \binom{N}{k} B^k (1-B)^{N-k} \tag{11}$$

where

$$F = \sum_{k=0}^{Q} \binom{N}{k} B^k (1-B)^{N-k}$$

And *N* random test cases are executed with *Q* failures, one has the confidence *C* that the true failure rate is no more than *B*. In other words, with a probability of at least *C*, one will see more than *Q* failures in *N* test cases when the failure density is more than *B*. Table 10 shows some computation results.
This model has the following characteristics:

1. Confidence value is between 0 and 1.
2. The maximum confidence from a given set of *N* test cases is obtained when there are no failures. By substituting *Q = 0*, one can obtain the original equation.
3. As the failure increases, the confidence decreases rapidly. When all test cases result in failures, the confidence is zero.
4. As the targeted failure density decreases, more test cases or fewer failures are required to achieve the same confidence.

Figure 24 shows how the confidence varies with the number of test cases for the target failure density 0.05, for the failures between *Q = 0* and *5*. Note that as the failures increases, the confidence decreases rapidly which is evident from the graphs becoming closer to the x-axis.

Table 10 Some combinations of *N, Q, B* and *C* when there are failures.

N	*Q*	*B*	*C*	*N*	*Q*	*B*	*C*	*N*	*Q*	*B*	*C*
50	0	0.01	0.39	50	2	0.01	0.01	50	5	0.01	0
50	0	0.05	0.92	50	2	0.05	0.46	50	5	0.05	0.04
100	0	0.01	0.63	100	2	0.01	0.08	100	5	0.01	0
100	0	0.05	0.99	100	2	0.05	0.88	100	5	0.05	0.38
250	0	0.01	0.92	250	2	0.01	0.58	250	5	0.01	0.04
250	0	0.05	0.99	250	2	0.05	0.99	250	5	0.05	0.99
400	0	0.01	0.98	400	2	0.01	0.76	400	5	0.01	0.21
400	0	0.05	0.99	400	2	0.05	0.99	400	5	0.05	0.99
600	0	0.01	0.99	600	2	0.01	0.94	600	5	0.01	0.56
600	0	0.05	0.99	600	2	0.05	0.99	600	5	0.05	0.99

When the failures increases from 0 to 5 out of 100 test cases, the confidence drops from 0.99 to around 0.4.

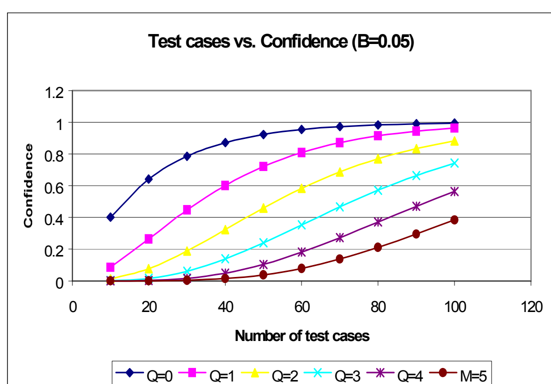

Fig. 24 Confidence in the Presence of Failures

In Fig. 25, for a fixed bound $B = 0.1$ the confidence is a function of failures detected for various number of test cases. For a 0.1 target failure rate with 0.95 confidence, the failures should be less than 5 out of 100 test cases. If the targeted confidence level is 0.8, the failures should be less than 7 out of 100 test cases. Note that reduction in the confidence does not increase the failures significantly. This is to be expected because the confidence decreases rapidly with each additional failure. Hence, in practice only few failures can be tolerated.

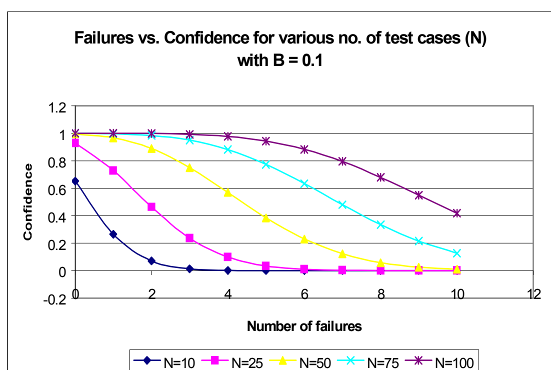

Fig. 25 Change in Confidence with Increasing Failures

Another important issue in using this new model is that the failures detected should not be critical. A single critical failure can disable mission-critical applications. Failures detected must be handled and tested after the SRBTAC process.

The SRBTAC end-to-end testing 3-Step Process

• Step 1: Run the regression testing. If the system fails at this step, it should be rejected; otherwise go to the next step.
– If the modified system cannot pass this step, the statistical models say that it is highly unlikely that the system will be able to pass the SRBTAC requirements.
– This step is relatively cheap because it reuses the existing test resources only.

Benefits and Experience of SRBTAC (II)

• It is relatively easy to identify which parts of subsystems are over tested and which are under tested or to prevent endless regresion test of defect detection and fixing loop of the feedback control model view for SDP-STP i.e. abnormality of planned test activities and established control limits .
• The testing team indicated that they can easily incorporate the SRBTAC requirements in their test projects if they were informed at the beginning of the project.
• The testing team indicated that it is easy to apply the SRBTAC process after some training.

7.3.3 Cost Means Risk

Is there a correlation between increasing the cost of testing and the ability to meet overall test goals? In our paper [17], we presented a model which showed that testing can be minimized by assessing the probability of successfully conducting the test based on cost. Analysis showed that for one-shot test events, such as bullets, bombs and missiles, the more expensive the test, the cost (in terms of achieving overall test goals) of failure (poor test attempt or failed test) increased. Costs are weighted by the probability of the cost being incurred by failing a test. These "costs" are not only the direct cost of the test itself (assets, range time, fuel, etc.) but also living with the results. Generally speaking the cost of incorrect evaluation (passing when it should be failed, or failing when it should be passed) exceeds the cost of correct evaluation (passing when it should be passed, or failing when it should be failed). The probability model analyzed past test history to determine if testing should continue, stop

with system pass, or stop with system failure. Figure 26 graphically depicts a generalized output of the behavior of the *OptimalSQM* framework in advanced quantitative defect control model used in the analysis. The regions are defined as follows: **Pass** - Indicates that one should quit testing and pass the system, **Test** - Indicates that one should test further, **Fail** - Indicates that one should quit testing and fail the system. It was shown that the Test (uncertain) region narrowed and shifted to the left as test costs increased and that the region of Pass (acceptance) region decreased. The Test region also necks down as the number of tests increases. As the cost per test is raised, it ultimately becomes too expensive to test the system and one uses what data is on hand to assess the system. Conversely as the cost of the test is lowered, one could test to system pass or quit and declare success earlier and minimize total expected cost. The author recommended that a test manager use a probabilistic based approach to minimize the expected total cost rather than to some fixed statistical pass threshold criteria. It can thus be shown analytically that a more costly test increases the risk of not meeting test requirements.

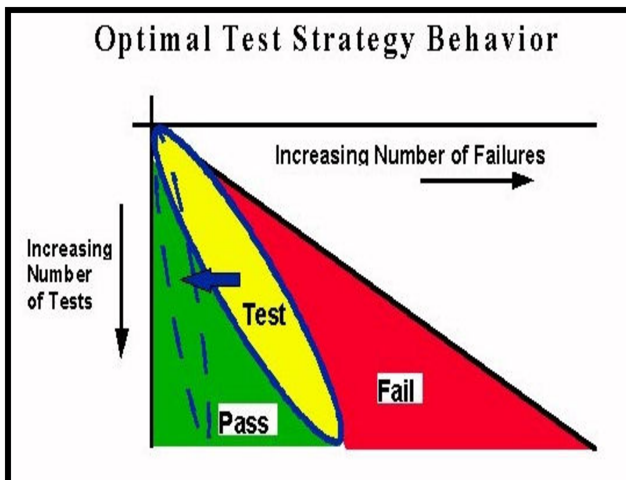

Fig. 26 Narrowing and shifting of the "Continue Test"

5 Conclusion

During the work on this project[1] several research questions were formulated which the research then was based upon. The initial main research question that was posed for the complete research in this project was: How can software testing be performed efficiently and effectively i.e. **Optimal**, that is, do we have a framework model targeted specific software testing domains or problem classes described in the paper? To be able to address the main research question several other research questions needed to be answered first (**RQ2–RQ5**). Thus, since this project is based upon the main research question, it was worthwhile taking the time to examine the current practice in different projects and see how software quality is measured and, especially, software testing was practiced [1-8] as we described in Section 2. In Section 3 and 4 we described our *OptimalSQM* framework which presents a set of best practice models and techniques integrated in optimized and quantitatively managed software testing process (*OptimalSQM*), expanding testing throughout the SDLC. To put it short, the answer to **RQ2** divided the research, as presented in this paper, into two areas:(1) covering effectiveness in software testing techniques, and (2) efficiency in software testing. We described how to implement development-testing alignment (DTA) methodology into *OptimalSQM* [5-9] which posits that such alignment leads to beneficial effects such as lower costs and shorter time of development, greater system quality, fewer errors and a better relationship between the corporate IT unit and customers in business functions who have commissioned new systems. To begin with, the research aimed at exploring the factor of defect detection and removing effectiveness DRE during SDLC (**RQ3**) while later focusing on early aspects of software cost of quality. In Section 6, we explained how can Advanced Quantitative Defect Management (AQDM) Model be enhanced (as answer to **RQ4**) is practically useful for determining which activities need to be addressed to improve the degree of early and cost-effective software fault detection. To enable software designers to achieve a higher quality for their design, a better insight into quality predictions for their design choices, test plans improvement using Simulated Defect Removal Cost Savings model is offered in paper. The model which enables to minimize the cost of switching between test plan alternatives, when the current choice cannot fulfill the quality constraints, corresponding optimality and stability criteria are proposed. Much rather we aim to define a simulation method with which it is possible to assist the test manager in evaluating test plan alternatives and adjusting test process improvement decisions in a systematic manner.To be practically useful for determining which activities need to be addressed to improve the degree of early and cost-effective software fault detection with assured confidence, than definitely, **optimality** and **stability** criteria of very complex STP dynamics problem control is described in Section 7 as answer to **RQ5**.

References:

[1] D. Galin, *Software Quality Assurance:From theory to implementation*, Pearson Education Limited, ISBN 0201 70945 7, 2004.

[2] A. Frost and M. Campo, " Advancing Defect Containment to Quantitative Defect Man", *CrossTalk*, December 2007.

[3] S. H. Kan, *Metrics and Models in Software Quality Engineering*, Second Edition, Addison-Wesley, 2003.

[4] C. Jones, *Estimating Software Costs*. 2nd edition. McGraw-Hill, New York: 2007.

[5] Lj. Lazić, N. Mastorakis, Cost Effective Software Test Metrics, *WSEAS TRANSACTIONS on COMPUTERS* , Issue 6, Volume 7, June 2008.

[6] Lj. Lazić, N. Mastorakis, Orthogonal Array application for optimal combination of software defect detection techniques choices, *WSEAS TRANSACTIONS on COMPUTERS*, Issue 8, Volume 7, August 2008.

[7] Lj. Lazić, N. Mastorakis. "Optimizing Test Process Action Plans by Simulated Defect Removal Cost Savings", 11th WSEAS Int.Conf. on AUTOMATIC CONTROL, MODELLING & SIMULATION (ACMOS'09), Istanbul, Turkey, May 30 - June 1, 2009.

[8] Lj. Lazićj. *The Integrated and Optimized Software Testing Process*. PhD Thesis, School of Electrical Engineering, Belgrade, Serbia, 2007.

[9] Lars-Ola Damm, Early and Cost-Effective Software Fault Detection, PhD Thesis, Blekinge Institute of Technology, SWEDEN, 2007.

[10] S. McConnell, *Professional Software Development*, Addison Wesley, ISBN 0-321-19367-9, 2004.

[11] B. Boehm, C. Abts, A. Brown, S. Chulani, B. Clark, E. Horowitz, R. Madachy, D. Reifer and B. Steece Software Cost Estimation with COCOMO II, Prentice Hall, 2000.

[12] Cohen, C. F., Birkin, S. J., Garfield, M. J. and Webb, H. W."Management Conflict in Software Testing." Communications of the ACM, 47(1), 76-81, 2004.

[13] Dhaliwal, J., Onita C., A framework for aligning Testing and Development, Proceedings of the Workshop on Advances & Innovations in Systems Testing, 2007.

[14] Hunter and Blosch "Managing the New IT Risks". Gartner, 2003.

[15] Pettichord, B. "Testers and Developers Think Differently: Understanding and Utilizing the Diverse Traits of Key Players on your Team." Software Testing & Quality Engineering, 2(1), 42-45, 2000.

[16] Sabherwal, Hirschheim and Goles. "Information systems – business strategy alignment: The dynamics of alignment: Insights form a punctuated equilibrium model. Strategic information management: Challenges and strategies in managing information systems", (Galliers and Leidner, Eds) pp 311-346, Butterworh-Heinemann, Oxford, 2003.

[17] Lj. Lazić Lj, D. Velasević. "Applying simulation and design of experiments to the embedded software testing process". STVR, Volume 14, Issue 4, p257-282, John Willey & Sons, Ltd., 2004.

[18] Lj. Lazić, N. Mastorakis. "The COTECOMO: COnstractive Test Effort COst Model". *SPRINGER VERLAG*, Proceedings of the European Computing Conference, Volume 2, Series: Lecture Notes in Electrical Engineering, Vol. 27 , Mastorakis, Nikos; Mladenov, Valeri (Eds.), ISBN: 978-0-387-84813-6, June 2009;89-110.

[19] Lj. Lazić, A. Kolašinac, Dž. Avdić. "The Software Quality Economics Model for Software Project Optimization", *WSEAS TRANSACTIONS on COMPUTERS,* Issue 1, Volume 8, p21-47, January 2009.

[20] M. Azuma. "SQuaRE: the next generation of the ISO/IEC 9126 and 14598 international standards series on software product quality". In ESCOM (European Software Control and Metrics conference), April 2001.

[21] R. S. Pressman, *Software engineering: a practitioner's approach* —5th ed., McGraw-Hill series in computer sc ience, 2001.

[22] R. Black, Managing the Testing Process, Second Edition. Wiley, New York, 2002.

[23] D. Houston, and B. Keats, "Cost of Software Quality: A Means of Promoting Software Process Improvement", Quality Engineering, 10:3, pp. 563-573, March, 1998.

[24] M. Müller, "About the Return on Investment of Test-Driven Development", ICSE'03, Portland, Oregon, 2003.

[25] Chrissis MB, Konrad M, Shrum S. "CMMI(R): Guidelines for Process Integration and Product Improvement". Addison-Wesley Publishing Company: Boston, MA. 2006.

[26] Q. Wang et al."Estimating Fixing Effort and Schedule based on Defect Injection Distribution", *Softw. Process Improve. Pract.*, 2008; 13: 35–50