Cost Effective Software Test Metrics

LJUBOMIR LAZIC[a], NIKOS MASTORAKIS[b]
[a]Technical Faculty, University of Novi Pazar
Vuka Karadžića bb, 36300 Novi Pazar, SERBIA
llazic@np.ac.yu http://www.np.ac.yu

[b]Military Institutions of University Education, Hellenic Naval Academy
Terma Hatzikyriakou, 18539, Piraeu, Greece
mastor@ieee.org)

Abstract: - This paper discusses software test metrics and their ability to show objective evidence necessary to make process improvements in a development organization. When used properly, test metrics assist in the improvement of the software development process by providing pragmatic, objective evidence of process change initiatives. This paper also describes several test metrics that can be implemented, a method for creating a practical approach to tracking & interpreting the metrics, and illustrates one organization's use of test metrics to prove the effectiveness of process changes. Also, this paper provides the Balanced Productivity Metrics (BPM) strategy and approach in order to design and produce useful project metrics from basic test planning and defect data. Software test metrics is a useful for test managers, which aids in precise estimation of project effort, addresses the interests of metric group, software managers of the software organization who are interested in estimating software test effort and improve both development and testing processes.

Key-Words: - Software testing, Test metrics, Size estimation, Effort estimation, Test effectiveness evaluation

1 Introduction

As organizations strive to shorten the development time of their products while at the same time attempting to improve their quality, the need for practical, cost effective testing strategies and techniques is becoming more and more important [1,2,3]. These strategies and techniques must span the full range of the development process addressing unit and component testing, integration testing, system testing and acceptance testing. In addition, the strategies and techniques must be tailored to the product under development recognizing unique project characteristics and constraints such as reliability, safety, cost and schedule. Testing activities also provide a critical opportunity to capture metrics and defect information that can be utilized to improve both development and testing processes. Software testing is one activity that can provide visibility into product and process quality. Test metrics are among the "facts" that project managers can use to understand their current position and prioritise their activities, so that they can reduce the risk (or impact) of running out of time before the software is ready for release [1,3-5].

We had many areas to address in our testing team to improve the quality and efficiency of the testing services provided to the project. Creating test metrics to drive process improvement in the test group and project team was a logical place to start. With metrics collection, timing is everything. Testers should start collecting metrics as soon as they get software that is stable enough to meaningfully run tests. If you are only testing (or collecting test metrics) near the end of the development lifecycle, then it is too late~ you have lost the opportunity to use the information to make a difference [4,6].

Test metrics collection programs do not have to be extensive to be effective. We have identified six issues (see Table 1) of test metrics that we collect on our testing projects. These fall into two categories – problem report (PR) information and test information. Testing is often seen as a troublesome and uncontrollable process. As it is often performed, it takes too much time, costs too much, and does not contribute to product quality. However, with appropriate processes, it can be brought under control and can add significant value to the development process. Planning for testing on a software project is often challenging for program managers. Test progress is frequently unpredictable, and during software testing painful schedule and feature "surprises" typically occur. Software testing is often viewed as an obstacle - more as a problem and less as a vital step in the process. For this reason, testing is treated as a "black box" and addressed at the end of the schedule. While budget

and time may be allocated for it, testing is not really managed in the same way as development.Typically, software development is measured in terms of overall progress in meeting functional and business goals. Software testing needs to be measured in similar terms to understand its true progress and make informed decisions. By considering testing dimensions other than cost and schedule, managers and other team members can better understand and optimize the testing process [3], in effect opening the black box and managing testing more effectively. We describe the Balanced Productivity Metrics (BPM) strategy and approach which incorporates the use of both quantitative and qualitative data for measuring performance and productivity improvement. In this way they can avoid costly and painful "surprises" late in the project. It is often said, "You cannot improve what you cannot measure." In this article, we describe some basic software measurement principles and suggest some metrics that can help you understand and improve the way your organization operates i.e. Software Testing Metrics Framework (STMF) [5]. When used properly, test metrics assist in the improvement of the software development process by providing pragmatic, objective evidence of process change initiatives. This paper also describes several test metrics that can be implemented, a method for creating and interpreting the metrics, and illustrates one organization's use of test metrics to prove the effectiveness of process changes.

Effective test management requires a wide variety of skills and activities, including the identification, collection, and analysis of a variety of test-related and quality-related metrics, and metrics associated with test status tracking, management, and control; proper reviews (to varying levels of formality) of test documentation and support material; and the determination of clear criteria for objectively assessing whether or not a system is ready for piloting, and when it is ready for operational use.Within this context, test activities should be prioritized with the ultimate objective of delivering maximum benefit to the end-users.).

This paper is organized as follows. Section 2 presents the software metrics definition. Section 3 explain why metrics specific to SW Testing are essential. Software Testing Optimization Model and IT benefits are presented in Section 4. Economic value measurement as a leading indicators for software testing process optimization is described in section 5. Finally, the paper is concluded in Section 6.

2 Software Metrics Definition

Metrics are defined as "standards of measurement" and have long been used in the IT industry to indicate a method of gauging the effectiveness and efficiency of a particular activity within a project. Test metrics exist in a variety of forms. The question is not whether we should use them, but rather, which ones should we use. Simpler is almost always better. For example, it may be interesting to derive the Binomial Probability Mass Function for a particular project, although it may not be practical in terms of the resources and time required to capture and analyze the data. Furthermore, the resulting information may not be meaningful or useful to the current effort of process improvement.

One thing that makes Test Metrics unique, in a way, is that they are gathered during the test effort (towards the end of the SDLC), and can provide measurements of many different activities that have been performed throughout the project. Because of this attribute, Test Metrics can be used in conjunction with Root Cause Analysis to quantitatively track issues from points of occurrence throughout the development process. Finally, when Test Metrics data is accumulated, updated and reported on a consistent basis, it allows trend analysis to be performed on the information, which is useful in observing the effect of process changes over multiple projects. Metrics are measurements, collections of data about project activities, resources and deliverables. Metrics can be used to help estimate projects, measure project progress and performance, and quantify product attributes. Examples include:

- product metrics, e.g., number of lines of code in a product, number of requirements in an SRS
- software development resource metrics, e.g., number of people working on a project
- software development process metrics, e.g., number of lines of code inspected

Metrics are measurements of the world around us. Without measurements, we are blind to the changes that go on in the world. Without measurements we can never know if we are improving or getting worse; we can never know if we are succeeding or failing. During a software development project, metrics have four basic uses;

1) to show the project manager where the project is in terms of completion,

2) to provide information upon which to base decisions,

3) to provide the basis of estimates for future projects, and

4) to provide management with information about the quality and reliability of the finished product.

In order to measure the actual values such as software size, defect density, verification effectiveness and productivity, records of these values must be maintained. Ideally, these actual values will be tracked against estimates that are made at the start of a project and updated during project execution.

The sole purpose for collecting metrics is to act on them. Failure to act on the data collected is waste. Actually, it is worse than waste because if the development community sees that the collected data is not being used, they will slowly stop collecting it or will report inaccurate values. This will reduce the waste but will render a metrics program useless and very difficult to restart. So if there is no written plan dealing with how the data is to be used, forget about collecting it.

2.1 Testing Process Flow

Once it was clear that Testing was much more than "Debugging" or "Problem fixing", it was apparent that testing was more than just a phase near the end of the development cycle. Testing has a lif e cycle of its own and there is useful and constructive testing to be done throughout the entire life cycle of development. This means that testing process begins with the requirements phase and from there parallels the entire development process. In other words, for each phase of the development process there is an important testing activity. This necessitates the need to migrate from an immature, adhoc way of working to having a full-fledged Testing Process. The following is the life cycle for the complete Test Development and Execution Process scheme (see Fig. 1).

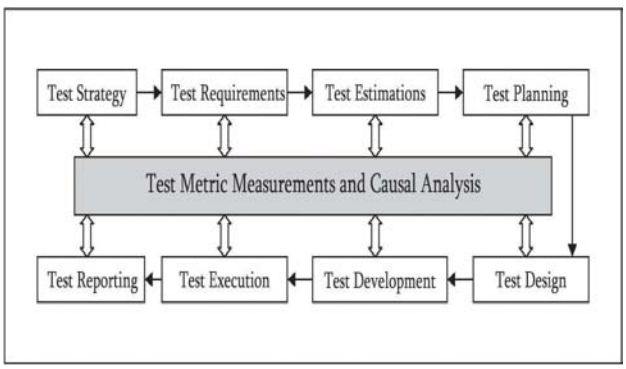

Fig. 1. Test Development and Execution Process scheme.

2.2 The New Test Metrics Philosophy

An optimized IT organization balances quality, cost and schedule. In doing so, IT can prioritize testing activities, make effective use of limited resources and be more agile in response to business change as described in our well documented IOSTP [3]. Test metrics and data gathering regarding the testing costs, testing failure costs, and defects are essential to manage and control testing function efficiently and effectively by a comprehensive Metrics Program that we call Software Testing Metrics Framework (STMF) [5] which is established and maintained to periodically check the health of the Testing Process with respect to "Defect Detection" and "Defect Prevention" effectiveness. This is done by "Monitoring & Measuring" the different metrics associated with "Defect Detection" and "Defect Prevention". Whenever, or wherever, the Testing process is found to be ineffective, it is "Optimized" accordingly [3,4]. Accurate data and relevant metrics provide information for decision making in relation to quality of products and processes. Otherwise the release decisions, further investments, and process changes are troublesome to justify without proper information. Hard data about the current situation also concretizes the true facts enabling to set up feasible and rational objectives. By establishing appropriate metrics, an organization can balance the cost of testing with the benefits derived from that testing. In order for metrics to be effective, the data collected must allow an organization to understand clearly:

• When the cost of further testing would outweigh the risk to the business [1,4].

• The cost to fix defects at the various stages of a project life cycle [6-8].

• The potential risk and subsequent costs to the business if the amount of testing were to be reduced [3].

This information can then be used to provide the organization with an informed basis of decision and effective ways to:

• Estimate the testing budget/spend.

• Spend more efficiently for future projects.

• Potentially reduce the overall costs of testing, realizing maximum value.

• Reduce total development and production support costs.

During individual projects, project metrics can be compared with accumulated experience to provide an early indication of quality levels and the accuracy of estimates. This in turn enables effective management and cost control at a project management level. In most organizations there is a lot of Do-Do-Do-Do and in many organizations there is a lot of Plan-Do-Plan-Do. But to close the cycle, the other two activities must be added [3]. We plan (make estimates), Do (execute our plan), Measure (measure our progress), and Act (Compare the actual progress against the estimated progress and make changes to reduce the difference) i.e. we implemented Six Sigma strategy to software testing process. For beginners at software metrics, this cycle can be applied to cost, effort and schedule, given the right measurements. The encompassing body is the Software Development Life Cycle (SDLC). At the beginning of the project, the key parameters like "Schedule", "Cost" and "Quality" are "Estimated" or "Predicted". These are subsequently monitored throughout the life cycle. The "Schedule" is monitored in terms of "Time Slip". The "Cost" is monitored in terms of "Effort Slip". The "Quality" is monitored in terms of "Defect Density". Testing Process, as can be seen from the above, is a sub set of the software development life cycle. The main focus areas are:

• Defect Detection

• Defect Prevention

as shown on Fig 2.

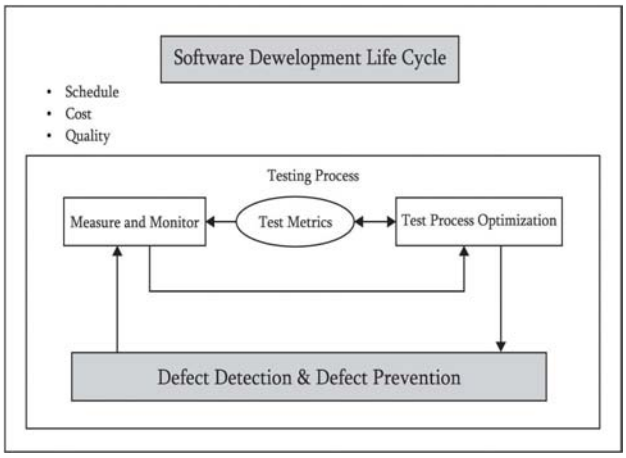

Fig. 2. The Software Testing Metrics Framework Process scheme.

2.3 Keep it Simple

What Are Software Metrics?

Software metrics are measures that are used to quantify software, software development resources, and/or the software development process. This includes items which are directly measurable, such as lines of code, as well as items which are calculated from measurements, such as earned value. Everyone who develops software uses some kind of software metrics. However, when asked what software metrics are, the tendency is to restrict the response to software size measurements, such as lines of code or function points. In reality, software metrics include much more than primitive measures of program size. Software metrics include calculations based on measurements of any or all components of software development. For example, consider the system integrator who wishes to determine the status of a project's test phase. He or she will undoubtedly ask for information on the proportion of tests that have been executed, the proportion that were executed successfully, and the number of defects identified. These measures are all examples of primitive - yet useful - software metrics. Consider the engineer who is responsible for improving the performance of a software product. He or she will consider items such as memory utilization, I/O rates, and the relative complexity of software components. These are also examples of software metrics. There is nothing overly complicated about software metrics. In fact, the biggest challenge in establishing an effective metrics program has nothing to do with the formulas, statistics, and complex analysis that are often associated with metrics. Rather, the difficulty lies in determining which metrics are valuable to the company, and which procedures are most efficient for collecting and using these metrics.

Our's STMF-Metrics methodology [5] begins by showing the value of tracking the easy metrics first. So, what are "easy metrics"? Most Test Analysts are required to know the number of test cases they will execute, the current state of each test case (Executed/Unexecuted, Passed/Failed/Blocked, etc.), and the time and date of execution. This is basic information that is generally tracked in some way by every Test Analyst. When we say "keep it simple" we also mean that the Metrics should be easy to understand and objectively quantifiable. Metrics are easy to understand when they have clear, unambiguous definitions and explanations.

Below are some examples of the definition and explanation of the 'easy metrics'.

2.4 Create Meaningful Metrics

Test Metrics are meaningful if they provide objective feedback to the Project Team regarding any of the development processes - from analysis, to coding, to testing – and support a project goal. If a metric does not support a project goal, then there is no reason to track it – it is meaningless to the organization. Tracking meaningless metrics wastes time and does little to improve the development process. Metrics should also be objective. As indicated in the sample definition shown in the previous section, an objective metric can only be tracked one way, no matter who is doing the tracking. This prevents the data from being corrupted and makes it easier for the project team to trust the information and analysis resulting from the metrics. While it would be best if all metrics were objective, this may be an unrealistic expectation. The problem is that subjective metrics can be difficult to track and interpret on a consistent basis, and team members may not trust them. Without trust, objectivity, and solid reasoning, which is provided by the Test Metrics, it is difficult to implement process changes. Metrics are important in Software because so much is at stake. Our jobs and the jobs of others depend on the cost of producing and maintaining software as well as the quality costs that may be incurred by us as the producers and by the users. If our competitors do a better job in producing a similar product, then our company will lose sales and we may lose our jobs. The very existence of a software producing organization may hang is the balance. Metrics can tell us how well we are doing and where we can improve by doing something different.

2.5 Track the Metrics

Tracking Test Metrics throughout the test effort is extremely important because it allows the Project Team to see developing trends, and provides a historical perspective at the end of the project. Tracking metrics requires effort, but that effort can be minimized through the simple automation of the Run Log (by using a spreadsheet or a database) or through customized reports from a test management or defect tracking system. This underscores the 'Keep It Simple' part of the philosophy, in that metrics should be simple to track, and simple to understand. The process of tracking test metrics should not create a burden on the Test Team or Test Lead; otherwise it is likely that the metrics will not be tracked and valuable information will be lost. Furthermore, by automating the process by which the metrics are tracked it is less likely that human error or bias can be introduced into the metrics.

2.6 Types of Metrics – Basic and Calculated

Basic metrics constitute the raw data gathered by a Test Analyst throughout the testing effort. These metrics are used to provide project status reports to the Test Lead and Project Manager; they also feed into the formulas used to derive Calculated Metrics. We suggests that every project should track the following Test Metrics:

# of Test Cases	# of First Run Failures
# of Test Cases Executed	Total Executions
# of Test Cases Passed	Total Passes
# of Test Cases Failed	Total Failures
# of Test Cases Under Investigation	Test Case Execution Time
# of Test Cases Blocked	Test Execution Time
# of Test Cases Re-executed	

As seen in the ‘Keep It Simple’ section, many of the Basic Metrics are simple counts that most Test Analysts already track in one form or another. While there are other Basic Metrics that could be tracked, we believes this list is sufficient for most Test Teams that are starting a Test Metrics program. Calculated Metrics convert the Basic Metrics data into more useful information. These types of metrics are generally the responsibility of the Test Lead and can be tracked at many different levels (by module, tester, or project). The following Calculated Metrics are recommended for implementation in all test efforts.

% Complete	% Defects Corrected
% Test Coverage	% Rework
% Test Cases Passed	% Test Effectiveness
% Test Cases Blocked	% Test Efficiency
1st Run Fail Rate	Defect Discovery Rate
Overall Fail Rate	Defect Removal Cost

These metrics provide valuable information that, when used and interpreted, oftentimes leads to significant improvements in the overall SDLC. For example, the 1st Run Fail Rate, as defined in the STMF- Metrics Methodology, indicates how clean the code is when it is delivered to the Test Team. If this metric has a high value, it may be indicative of a lack of unit testing or code peer review during the coding phase. With this information, as well as any other relevant information available to the Project Team, the Project Team may decide to institute some preventative QA techniques that they believe will improve the process. Of course, in the next project, when the metric is observed it should be noted how it has trended to see if the process change was in fact an improvement.

2.7 The final step - Interpretation and Change

As mentioned earlier, test metrics should be reviewed and interpreted on a regular basis throughout the test effort and particularly after the application is released into production. During the review meetings, the Project Team should closely examine ALL available data, and use that information to determine the root cause of identified problems. It is important to look at several of the Basic Metrics and Calculated Metrics in conjunction with one another, as this will allow the Project Team to have a clearer picture of what took place during the test effort. If metrics have been gathered across several projects, then a comparison should be done between the results of the current project and the average or baseline results from the other projects. This makes trends across the projects easy to see, particularly when development process changes are being implemented. Always take note to determine if the current metrics are typical of software projects in your organization. If not, observe if the change is positive or negative, and then follow up by doing a root cause analysis to ascertain the reason for the change.

3 The Metrics Specific to SW Testing are Essential

Metrics help you better control your software projects and learn more about the way your organization works. Specifically, the measurements described in this paper first answers the question of whether Software Testing is "doing the right thing" (effectiveness). Once there is assurance and quantification of correct testing, metrics should be developed that determine whether or not Software Testing "does the thing right" (efficiency) as we did during M&S of Optimized Software Testing model which combine Risk Management and Earned Value Management called RBOST [4]. You can measure many aspects of your software products, projects, and processes. The trick is to select a small and balanced set of metrics that will help your organization track progress toward its goals. Major components (depicted in Fig. 3) of proposed Software Testing Metrics Framework are: 1) The Goal Question Metric (GQM) process, created by Victor Basili and his colleagues at the University of Maryland [2], is a good place to begin targeting the specific measurement needs of an organization, 2) Balanced Scorecard (BSC) that ensures set of measures providing coverage of all elements of performance and avoid hidden trade-offs and 3) Process Model Performance measures that are most meaningful with respect to selected areas of performance, prefere outcome then output measures over process and input measures. The main emphasis of GQM is goal directed measurement. An organization usually starts with generic goals that must be refined. For example, “Reduce the number of failures found on a project”. This is certainly a goal, but is it well enough refined? One technique to further refine goals, making them specific enough that they are applicable to the direction of the organization, is the SMART technique.

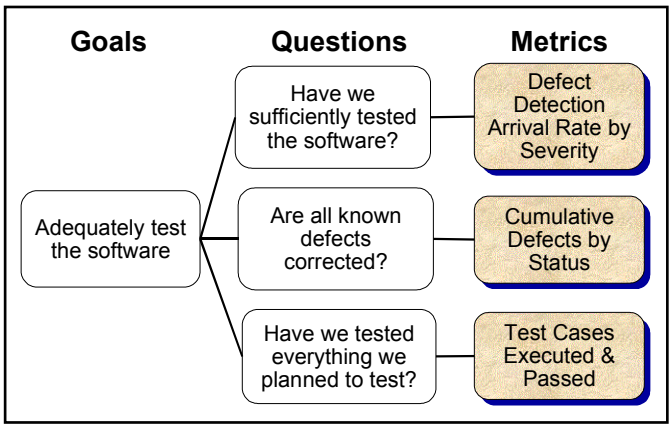

Fig. 3. The Goal / Question / Metrics cheme in STMF

To answer these questions, specific categories of measurement data must be available to the project manager. The issues, key questions related to each issue, and categories of measures necessary to answer the questions are show in Table 1. Furthermore, we need to consider the efficiency of the test effort that is a part of the process and a determinate of reliability and risk of deployment. The relationship between product quality and process capability and maturity has been recognized

as a major issue in software engineering based on the premise that improvements in process will lead to higher quality products. To this end, we have been investigating an important facet of process capability – stability – as defined and evaluated by trend, change, and shape metrics, across releases and within a release. Our integration of product and process measurement serves the dual purpose of using metrics to assess and predict reliability and risk and to evaluate process stability [3].

3.1 Basic Software Testing Process Metrics

By focusing data collection activities on measurement categories that answer the key issue questions the project can minimize resources devoted to the measurement process. Among many Goals and Problems identified in former SDP/STP, before IOSTP deployment [3], our focus for STP improvement for demonstration purpose in this paper were issues - Development/Testing Performance and Product Quality i.e. only to these sampled issues, key questions related to each issue, and categories of measures necessary to answer the questions are show in Table 1 to 4 and some graphical presentations in figures 4 to 6. Measuring the impact and consequences of problems that arise during testing is a critical step in the process. This should include analysis of collected measurements and calculated metrics to find out how much of the software is affected by a given problem, at what point during testing a problem was found, and what kinds of problems regression tests are attempting to uncover.

Table 1. The issues, key questions related to each issue, and categories of measures

Issue	Key Questions	Measurement Category
1. Schedule & Progress	Is the project meeting scheduled milestones?	1.1 Milestone Performance
	How are specific activities and products progressing?	1.2 Work Unit Progress
	Is project spending meeting schedule goals?	1.3 Schedule Performance
	Is capability being delivered as scheduled?	1.4 Incremental Capability
2. Resources & Cost	Is effort being expended according to plan?	2.1 Effort Profile
	Are qualified staffs assigned according to plan?	2.2 Staff Profile
	Is project spending meeting budget objectives?	2.3 Cost Performance
	Are necessary facilities and equipment available as planned?	2.4 Environment Availability
3. Growth & Stability	Are the product size and content changing?	3.1 Product Size & Stability
	Are the functionality and requirements changing?	3.2 Functional Size & stability
	Is the target computer system adequate?	3.3 Target Computer Resource Utilization
4. Product Quality	Is the software good enough for delivery?	4.1 Defect Profile
	Is the software testable and maintainable?	4.2 Complexity
5. Development / Testing Performance	Will the developer be able to meet budget and schedules?	5.1 Process Maturity
	Is the developer efficient enough to meet current commitments?	5.2 Productivity
	How much breakage to changes and errors has to be handled?	5.3 Rework
6. Technical Adequacy	Is the planned impact of the leveraged technology being realized?	6.1 Technology Impacts

Table 2. Key questions related to each issue, and categories of measures

4. Product Quality	Is the software good enough for delivery?	4.1 Defect Profile
5. Development / Testing Performance	Is the developer efficient enough to meet current commitments?	5.2 Productivity

Once a list of valid questions are created, measurements are generated. When considering metrics, it is often helpful to list the raw data that must be collected. This raw data is sometimes referred to as "primitive metrics". In this example, some important raw data is:

- Number of critical defects with a severity level of three and four.
- Time in duration testing.
- Total number of defects found in duration testing time period.
- Number of critical defects found on the last project for the corresponding time period.
- Number of total defects on last project for the corresponding time period.

Table 3. Measurement Category and Specific Measures

Measurement Category	Specific Measures
4.1 Defect Profile	4.1.1 Problem Report Trends 4.1.2 Problem Report Aging 4.1.3 Defect Density 4.1.4 Failure Interval

Once the raw data is defined, more complex, or "computed" metrics are generated based on combinations of primitive metrics.

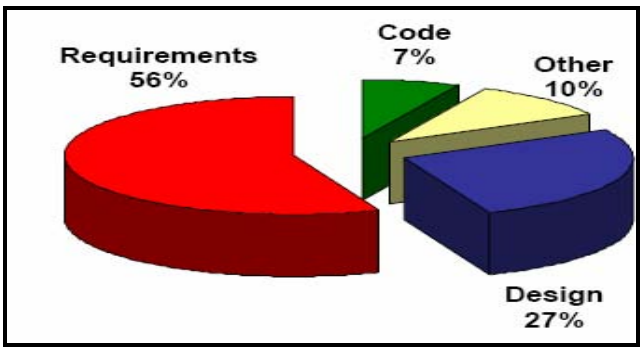

Fig. 4 Typical Distribution of Bugs

Deriving measurements from raw data and translating that data into something useful to managers and/or developers is essential in tracking real progress towards a goal. Important computed metrics in this example are:

- Number of critical failures found in duration testing time period / Total number of failures found in duration testing time period.

- Numb**e**r of critical failures (severity 3&4) found in corresponding time period on previous project/Total number of failures found in corresponding time period on previous project.

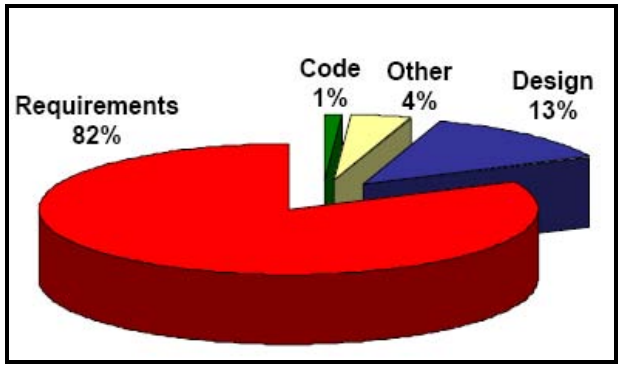

Fig. 5 Typical Distribution of Effort to Fix Bugs

After collection and analysis phase statistical methods and tools are used to identify and confirm root causes of defects. Not only must analysis of the data be performed, but also an in depth analysis of the process to ensure an understanding of how the work is actually being done must be performed to identify inconsistencies or problem areas that might cause or contribute to the problem. Deliverables of this phase are: data and process analysis, root cause analysis, quantifying the gap/opportunity and checkpoints for completion is to identify gaps between current performance and the goal performance.

Root Cause Analysis should be done to:

- Generate list of possible causes (sources of variation).

- Segment and stratify possible causes (sources of variation).
- Prioritize list of 'vital few' causes (key sources of variation).
- Verify and quantify the root causes of variation.

Status of Open Deficiencies in Delivered Projects

Severity Levels	Number of Deficiencies That Have Been Open x Days				Totals
	x < 30	30 < x ≥ 60	60 < x ≥ 90	x > 90	
Severity 1	2	1			3
Severity 2	3	1	1		5
Severity 3	3	2	1	1	7
Severity 4	4	3	3	2	12
Severity 5	8	6	3	3	20
Totals	20	13	8	6	47

Fig. 6 Typical time to Fix Bugs vs severity levels

In order to quantify the Gap/Opportunity answering the questions:

- What is the cost of poor quality as supported by the team's analysis?
- Is the process severely broken such that a re-design is necessary?
- What are the revised rough order estimates of the financial savings/opportunity for the improvement project?
- Have the problem and goal statements been updated to reflect the additional knowledge gained from the analyze phase?
- Have any additional benefits been identified that will result from closing all or most of the gaps?
- What were the financial benefits resulting from any 'ground fruit or low-hanging fruit' (quick fixes)?
- What quality tools were used to get through the analyze phase?

Table 4. Focus question and specific measure

4 PRODUCT QUALITY	
Are difficult problems being deferred?	**4.1.2 Problem Report Aging**
Are reported problems being closed in a timely manner?	**4.1.2 Problem Report Aging**
Do report arrival and closure rates support the scheduled completion date of integration and test?	**4.1.1 Problem Report Trends**
FOCUS QUESTION	**SPECIFIC MEASURE**
How long does it take to close a problem report?	**4.1.2 Problem Report Aging**
How many problem reports are open? What are their priorities?	**4.1.1 Problem Report Trends**
How many problems reports have been written?	**4.1.1 Problem Report Trends**
How much code is being reused?	**4.2.6 Depth Of Inheritance**
How often will software failures occur during operation of the system?	**4.1.4 Failure Interval**
How reliable is the software?	**4.1.4 Failure Interval**
What components are candidates for rework?	**4.1.3 Defect Density**
What components have a disproportionate amount of defects?	**4.1.3 Defect Density**
What components require additional testing or review?	**4.1.3 Defect Density**
What is the program's expected operational reliability?	**4.1.4 Failure Interval**
What is the quality of the software?	**4.1.3 Defect Density**

In proposed STMF our focus is on software Error and Defect Root Cases Analysis applying Defect Classification scheme as described in our paper about Software Testing Process Improvement to achieve a high ROI of 100:1 [5]. This information is needed to monitor the overall progress of the software through testing and to make informed decisions about software release. So by combining all of the different perspectives of schedule, functionality, code, and problem resolution, it is possible to understand and manage software testing, rather than treating it as a black box as we explained in our paper of proposed STMF, Part 2 [5].

3.1 Understanding Defects

Tracking defects means recording every defect found by a customer or by any formal testing process. That does not mean asking individuals to track every change – unit tests, individual code reviews and walkthroughs are usually not subject to tracking. Once an author declares a work product "complete" however, and releases it for independent appraisal by others, all defects found should be tracked.

Every defect found should be identified with the following information:

- Work product in which the defect was found. (This may need to be determined by the person doing the fix.)
- Defect Detection Technique (DDT) used to find the defect – inspection, type of testing, customer, etc.
- Origin of defect, i.e., where it was "inserted" – requirements, design, coding, etc. (It is not always possible to determine the origin, but an adequate sample is usually feasible.)
- Project ID and life cycle phase (if during the project) or application ID in which the defect was found (if after release).
- Other information necessary to track "assigned to," "status" and "closing information." (This is not necessarily needed for process improvement, but typically is required for management purposes.)
- Defect type – a short list of orthogonal categories that make it possible to determine which type of defect is most effectively found by which type of appraisal.

Tools such as a defect cost scorecard, defect containment effectiveness (DCE) and total containment effectiveness (TCE) metrics can be applied to manage differential effectiveness across phase of origin and detection.

The following indicators make a good effectiveness dashboard. They are intended to be used in three perspectives – baseline (values at the start), trend (changes in values over time, reflecting aggregate impact of all interventions) and pre/post intervention (reflecting the impact of a specific change or improvement under reasonably controlled conditions, making an effort to isolate individual effects).

- DDT cost per defect by phase and DDT type (by project and in aggregate)
- Rework cost per defect by phase and appraisal type (by project and in aggregate)
- Value-added, appraisal and rework as a percentage of effort (by project and in aggregate)
- Defect containment effectiveness (by project and in aggregate)
- Total containment effectiveness (by project and in aggregate)
- Effort variance normalized for size (by project and in aggregate)
- Schedule variance normalized for size (by project and in aggregate)
- Defect density, or defect per size – total "insertion" rate (by project and in aggregate)
- Effort per size, or productivity (essential to consider variations in "schedule pressure")
- Duration per size, or cycle time (essential to consider variations in "schedule pressure")

Based on experience with sustained application of these metrics, it is typically possible for an organization to shift 10 percent to 20 percent of non-value-added work to value-added within one to two years. It is not easy, but the payoff potential is very large.

By comparing defect counts and defect fix efforts from two completed software projects, we utilize a scorecard to summarize and highlight differences in actual defect counts by development phase of origin and the phase in which the defect was detected.

In software testing efficiency we refer to two calculated yields: 1) Phase Containment Effectiveness (PCE - the % of errors detected during the phase in which they were introduced). Those that escape the current phase are considered defects - we then calculate: 2) Defect Containment Effectiveness (DCE - the percent of the defects

escaping an earlier phase that are detected in the current phase).

Defects are real, observable manifestations and indications of the software development progress and process:

- From a schedule viewpoint – progress
- From a quality viewpoint – early indication
- From a process engineering – indicate effectiveness of different parts of the process and targets for improvement.

You can see them, count them, predict them, trend them with:

- *Defect Density:* a standard quality measure expressed in number of defects per KLOC or FP;
- Defect *Arrival Rates/Fix Rates*: a standard Process and Progress Measurements expressed in number of Defects detected/fixed per unit of time (or effort);
- *Injected* Defects: Defects which are put into the product (due to "errors" which people make) with sharacteristic:
    - When you have excellent processes, you have fewer injected defects;
    - You can never know completely, how many injected defects you have in your product – you can only estimate them;
- *Removed* Defects: a defects which are identified and then taken out of the product due to some defect removal activity (DDT), such as code reviews;
- Faults can range from crucial to inconsequential and must have a severity scheme that allows you to differentiate. Severity scheme needs to be based upon the project because we want to focus on defects that will actually impact your project and product performance;

Defects have certain dynamics, behaviors, and patterns which are important to understand in order to understand the dynamics of software development. In general projected Software Defects follow a Rayleigh Distribution Curve, so we can predict, based upon project size and past defect densities, the curve, along with the Upper and Lower Control Bounds.

3.2.1 Measurement of Defect Potentials and Defect Removal Efficiency

There are two very important measurements of software quality that are critical to the industry:

1. Defect potentials
2. Defect removal efficiency

All software managers and quality assurance personnel should be familiar with these measurements because they have the largest impact on software quality, cost, and schedule of any known measures.

The phrase *defect potentials* refers to the probable numbers of defects that will be found during the development of software applications. As of 2008, the approximate averages in the United States for defects in five categories, measured in terms of defects per function point and rounded slightly so that the cumulative results are an integer value for consistency with other publications by the author, follow.

Note that defect potentials should be measured with function points and not with lines of code. This is because most of the serious defects are not found in the code itself, but rather in requirements and design. Table 5 shows the averages for defect potentials in the U.S. circa 2008.

Requirements defects	1.00
Design defects	1.25
Coding defects	1.75
Documentation defects	0.60
Bad fixes	0.40
Total	**5.00**

Table 5 Averages for Defect Potential [6]

The measured range of defect potentials is from just below two defects per function point to about 10 defects per function point. Defect potentials correlate with application size. As application sizes increase, defect potentials also rise.

A useful approximation of the relationship between defect potentials and defect size is a simple rule of thumb: application function points raised to the 1.25 power will yield the approximate defect potential for software applications. Actually, this rule applies primarily to applications developed by organizations at Capability Maturity Model®(CMM®) Level 1. For the higher CMM levels, lower powers would occur. Reference [8] shows additional factors that affect the rule of thumb.

The phrase *defect removal efficiency* refers to the percentage of the defect potentials that will be removed before the software application is delivered to its users or customers. As of 2007, the average for defect removal efficiency in the U.S. was about 85 percent [7].

If the average defect potential is five bugs – or defects – per function point and removal efficiency is 85 percent, then the total number of delivered defects will be about 0.75 per function point. However, some forms of defects are harder to find and remove than others. For example, requirements defects and bad fixes are much more difficult to find and eliminate than coding defects.

At a more granular level, the defect removal efficiency against each of the five defect categories is approximate in Table 6.

Defect Origin	Defect Potential	Removal Efficiency	Defects Remaining
Requirements defects	1.00	77%	0.23
Design defects	1.25	85%	0.19
Coding defects	1.75	95%	0.09
Documentation defects	0.60	80%	0.12
Bad fixes	0.40	70%	0.12
Total	**5.00**	**85%**	**0.75**

Table 6 Defect Removal Efficiency

Note that the defects discussed in this section include all severity levels, ranging from severity 1 i.e. defects with the least impact, up to severity 5 i.e. defect at the most impact. Obviously, it is important to measure defect severity levels as well as recording numbers of defects.

There are large ranges in terms of both defect potentials and defect removal efficiency levels. The *best in class* organizations have defect potentials that are below 2.50 defects per function point coupled with defect removal efficiencies that top 95 percent across the board [6].

Defect removal efficiency levels peak at about 99.5 percent. In examining data from about 13,000 software projects over a period of 40 years, only two projects had zero defect reports in the first year after release. This is not to say that achieving a defect removal efficiency level of 100 percent is impossible, but it is certainly very rare [6].

Organizations with defect potentials higher than seven per function point coupled with defect removal efficiency levels of 75 percent or less can be viewed as exhibiting professional malpractice. In other words, their defect prevention and defect removal methods are below acceptable levels for professional software organizations [6].

As can be seen from the short discussions here, measuring defect potentials and defect removal efficiency provide the most effective known ways of evaluating various aspects of software quality control. The phrase *defect prevention* refers to technologies and methodologies that can lower defect potentials or reduce the numbers of bugs that must be eliminated. Examples of defect prevention methods include joint application design, structured design, and also participation in formal inspections.

The phrase *defect removal* refers to methods that can either raise the efficiency levels of specific forms of testing or raise the overall cumulative removal efficiency by adding additional kinds of review or test activity. Of course, both approaches are possible at the same time [3,6].

Since each testing stage will only be about 30 percent efficient, it is not feasible to achieve a defect removal efficiency level of 95 percent by means of testing alone. Formal inspections will not only remove most of the defects before testing begins, it also raises the efficiency level of each test stage. Inspections benefit testing because design inspections provide a more complete and accurate set of specifications from which to construct test cases [1,3].

From an economic standpoint, combining formal inspections and formal testing will be cheaper than testing by itself. Inspections and testing in concert will also yield shorter development schedules than testing alone. This is because when testing starts after inspections, almost 85 percent of the defects will already be gone. Therefore, testing schedules will be shortened by more than 45 percent [6-9]. Measuring the numbers of defects found during reviews, inspections, and testing is also straightforward. To complete the calculations for defect removal efficiency, customer-reported defect reports submitted during a fixed time period are compared against the internal defects found by the development team. The normal time period for calculating defect removal efficiency is 90 days after release.

As an example, if the development and testing teams found 900 defects before release, and customers reported 100 defects in the first three months of usage, it is apparent that the defect removal efficiency would be 90 percent.

3.3 The optimized testing approach

Adopting an optimized testing approach may sound overwhelming. However, the truth is that IT organizations can adopt optimized testing practices incrementally, implementing certain aspects tactically to achieve strategic advantage. This section offers some suggested ways to adopt an optimized testing approach. Optimized testing best

practices include adoption of the application life cycle, requirements management, risk-based testing and automation [4]. This section highlights some suggested steps for incorporating these areas of optimized testing into your existing testing environment.

3.3.2 The Defect-Removal Filtering Process

The cost of fixing defects increases exponentially as a defect moves through development into production. By adopting an application quality life cycle, quality is built in to the application from the earliest phases of its life cycle, rather than attempting to test it in when it's too late. This requires discipline in defining and managing requirements, implementing automated and repeatable best practices and access to the right information to make confident decisions. This best practice allows you to fix defects earlier, when there is more time to sufficiently address the problems and it is far less expensive. It lays the groundwork for continuous process improvement and higher-quality applications [3,8].

The specification defines the program correct behavior. The incorrect behavior is a software failure. It can be improper output, abnormal termination, and unmet time or space constraints. Failures are mostly caused by faults, which are missing or incorrect code. Error is a human action that produces a failure. An abend is an abort termination of a program (like "blue screen of death" by Microsoft Windows). An omission is a required capability, which is not present in an implementation. Surprise is code that does not support a required capability. It can be surprising at code reuse. Bug is an error or a fault. The scope of STP is the collections of artifacts under test (AUT). Testing activities can be categorized by the scope of AUT that belongs to corresponding STP or SDL phase as show on Fig. 7. Test artifact under test can be the software requirement (SRUT), High level design (HLDUT), Low Level Design (LLDUT), code being tested is called implementation under test (CUT), integration test (IUT) system under test (SUT), or in object-oriented environment class under test (CLUT), object under test (OUT), method under test (MUT). A test case defines the input sequence of data, the environment and state of AUT, and the expected result. Expected result is what AUT should generate, actual result is what was generated by run. An oracle produces expected results. An oracle can be an automated tool or human resource. A test is called to be passing if expected results and actual results are equal, otherwise it is called to be no pass or fail.

Test cases can be designed for positive testing or negative testing. Positive testing checks that the software does what it should. Negative testing checks that the software does not do what it should not do. A test suit is a collection of test cases related to each other. Test run is the execution of a test suit. Test driver is a tool (can be a unit or utility program) that applies test cases to AUT. A stub is a partial, temporary implementation of a component. The following figure shows the systems engineering view of testing. Test strategy identifies the levels of testing, the methods, test detection techniques (DDT) and tools to be used. Test strategy defines the algorithm to create test cases.

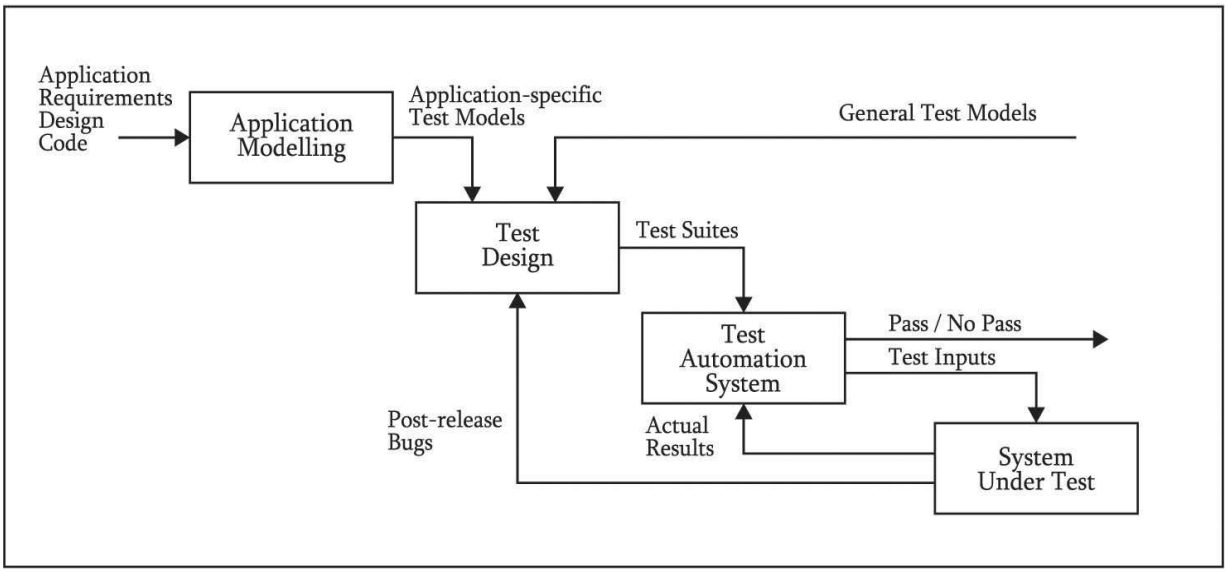

Fig. 7 The systems engineering view of testing

Test design produces test cases using a test strategy. Test effectiveness of DDT is the ability of the test strategy to find the bugs that we call the Defect-Removal Filtering Process as shown on Fig. 8. Test efficiency is the cost of finding bugs. Strategies for the test design can be functional (Black-box), structural (White-box), hybrid (Gray-box) and fault-based. Functional testing is based on the specification of software, without knowing something about program code. It uses the specified or expected behavior. It is also called specification based, behavioral, and responsibility-based or black-box testing.

Structural testing relies on the structure of the source code to develop test cases. It uses the actual implementation to create test suits. It is also called implementation based, white box or clear box testing. Hybrid testing is the blend of functional and structural testing. It is also called gray-box testing.

Fault-based testing introduces faults into code (mutation) to see if these faults are revealed by a test suite. Regression testing is retesting the software with the same test cases. After a bug is fixed, the product should be tested at least with the bug revealer test case.

Coverage is the percentage of elements required by a test strategy that have been exercised by a test suite. There are many coverage models. Statement coverage is the percentage of source code statements executed at least once by a test suite. Clearly, statement coverage can be used only by structural or hybrid testing. Testing should make effort to reach 100% code coverage. This can avoid the user to run untested code. All these definitions raise a lot of questions and problems, and all of them cannot be dealt in this article (see references in [3]), although the most important ones can be found below. The testing strategy defines how test design should produce the test cases, but nothing it can tell us about how much testing is enough, and how effective the testing was. Test case effectiveness depends on numerous factors, and can be evaluated after the end of testing, which is normally too late. To avoid these problems, testers should perform in-process evaluation of proposed and planned STP, according to established performance metrics and quality criteria [1,3] as we described below.

According to a National Institute of Standards and Technology (NIST) study, *the problem of continued delivery of bug-ridden software* is costing the U.S. economy an estimated $59.5 billion each year. The study also found the following [8]:

"…although all errors cannot be removed, more than a third of these costs, or an estimated $22.2 billion, could be eliminated by an improved testing infrastructure [reviews, inspections, etc.] that enables earlier and more effective identification and removal of software defects. These are the savings associated with finding an increased percentage [but not 100 percent] of errors closer to the development stages in which they were introduced. Currently, over half of all errors are not found until 'downstream' in the development process (testing) or during post-sales software use"

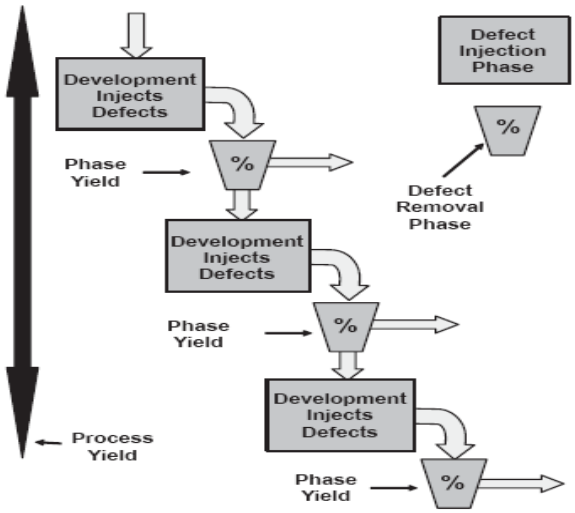

Fig. 8 The Defect-Removal Filtering Process

The testers should verify test cases at the end of test design, check the conformance of test cases to meet the requirements. It should also check the specification coverage. Validation is after test execution. Knowing the result, an effectiveness rate should count, and if it is under the threshold, the test suite should be analyzed, and the test process should be corrected.

A simple metric for effectiveness, which is only test suite dependent [1]. It is the ratio of bugs found by test cases (N_{tc}) to the total number of bugs (N_{tot}) reported during the test cycle (by test cases or by side effect):

$$TCE = 100 * N_{tc} / N_{tot} \, [\%] \quad (1)$$

This metric can evaluate effectiveness after a test cycle, which provides in-process feedback about the actual test suite effectiveness. To this metric a threshold value should create. This value is suggested to be about 75%, although it depends on the application.

When the *TCE* value is above the threshold, the test case can be said effective according to very useful model for dealing with defects as depicted on Fig. 9. If it is below, testers should correct the test plan, focusing on side effect bugs.

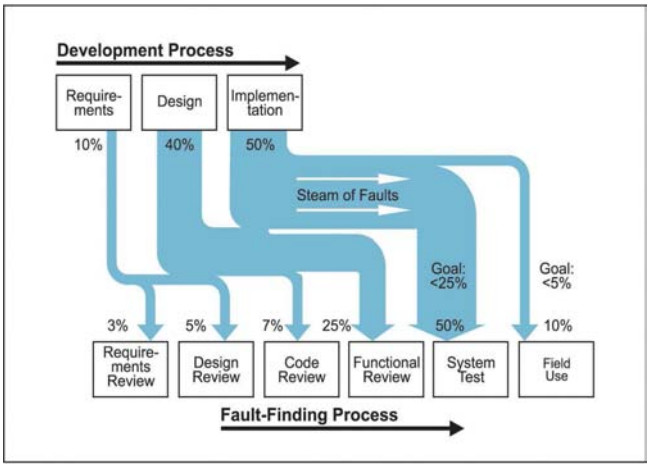

Fig. 9 Fault Injection Model Traditional

It basically says that given a software project – you have defects being "injected" into it (from a variety of sources) and defects being removed from it (by a variety of means). This high-level model is good to use to guide our thinking and reasoning about defects and defect processes. So, based on this model, the goal in software development, for delivering the fewest defects, is to: minimize the number of defects that go in maximize the number of defects that are removed.

3.3.3 Defect Removal Efficiency and Economics

A key metric for measuring and benchmarking the IOSTP [3] 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:

$$DRE = E/(E+D) \quad (2)$$

Where E is the number of errors found before delivery to the end user, and D is the number of errors found after delivery. The goal is to have *DRE close to 100%*. The same approach is applied to every test phase denoted wit i :

$$DRE_i = \frac{E_i}{(E_i + E_{i+1})} \quad (3)$$

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 DDT to achieve high DRE, let say >85%, as a threshold for STP required effectiveness, is explained in section 5. which describe optimum combination of software defect detection techniques choices determination applying *orthogonal arrays* constructed for post mortem designed experiment with collected defect data of a real project [3]. Figure 10 shows a typical relationship between the costs of repairing a defect in a given phase of the development cycle versus which phase the defect was introduced. This relationship gives rise to the evelopment costs described in the NIST report.

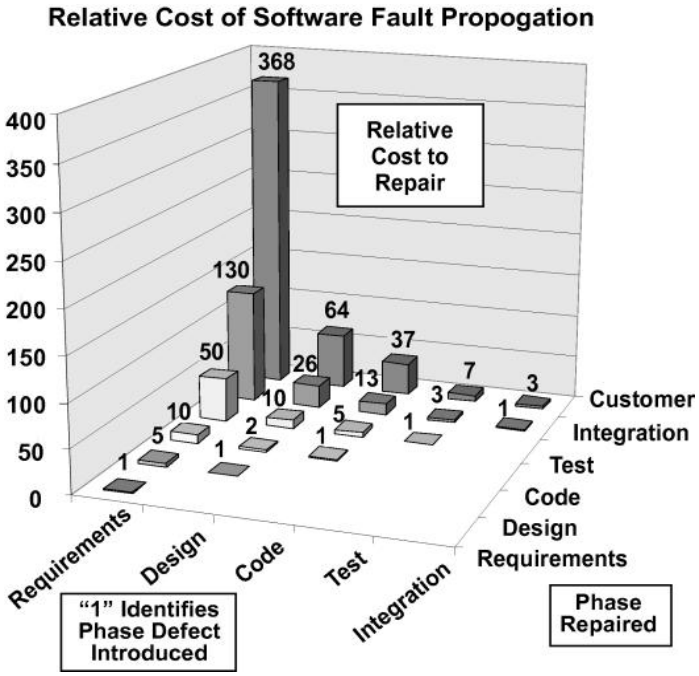

Fig. 10 Cost of Fixing a Defect [10]

Cost numbers vary depending on the type of application for which the software is being developed, but the common thread they all exhibit is the substantial increase in project costs caused by carrying problems from one development stage to the next.

While these numbers are extrapolated from software developed for the financial services and transportation applications (computer-aided design, computer-aided manufacturing, etc.) sectors, the message applies even more significantly to industries engaged in developing software for safety and mission-critical applications such as aerospace, medical, defense, automotive, etc. Failures of safety/mission-critical software may result in harm to, or loss of human life and/or mission objectives such as in the case of the Therac-25 radiation overdose accidents and the Ariane-5 maiden launch failure [10]. The Therac-25 software caused severe radiation burns in numerous cancer patients before it was implicated. The cost of allowing the Ariane-5 software defect to pass into the operational phase has been estimated to be as high as $5 billion alone.

NASA recently sponsored a study to evaluate the economic benefit of conducting independent validation and verification during the development of safety-critical embedded systems [10]. This study presented cost-to-repair figures focused specifically on embedded systems projects. Figure 10 shows the relative cost to repair factors – considered to be conservative estimates for embedded systems – used in this study.

The graph in Fig. 10 tells us that an error introduced in the requirements phase will cost five times more to correct in the design phase than in the phase in which it was introduced. Correspondingly, it will cost 10 times more to repair in the code phase, 50 times more in the test phase, 130 times more in the integration phase, and 368 times more when repaired during the operational phase. The graph also gives the cost multipliers for problems introduced in the design, code, test, and integration phases of the development cycle.

3.3.4 The ROI calculation and other benefits

To determine the cost savings for addressing these defects early in the software development lifecycle (shown in figure 11 below), we applied industry average defect-correction cost information to the number of critical defects discovered by IOSTP [3,4] during this project.

According to industry average data, the cost of finding and correcting defects during the *coding* phase is $977 per defect. Thus, the total cost for correcting the 200 "critical" defects during this phase (200 x $977) is approximately $195,400. Industry average data shows that the cost of finding and correcting defects during the *system testing* phase is $7,136 per defect. In this case, assuming that the system testing phase revealed approximately 50 critical defects (or only 25% of those found by IOSTP [3,4] in the coding phase), the cost of finding and fixing those defects (50 x $7,136) would have been approximately $356,800. This would also have resulted in 150 critical errors going undetected and uncorrected. The cost of finding and fixing these remaining 150 defects in the maintenance phase (150 x $14,102) would have been $2,115,300. Thus, the total cost of finding and fixing the 200 defects *after* the coding phase would have been $2,472,100 ($2,115,300 + $356,800)

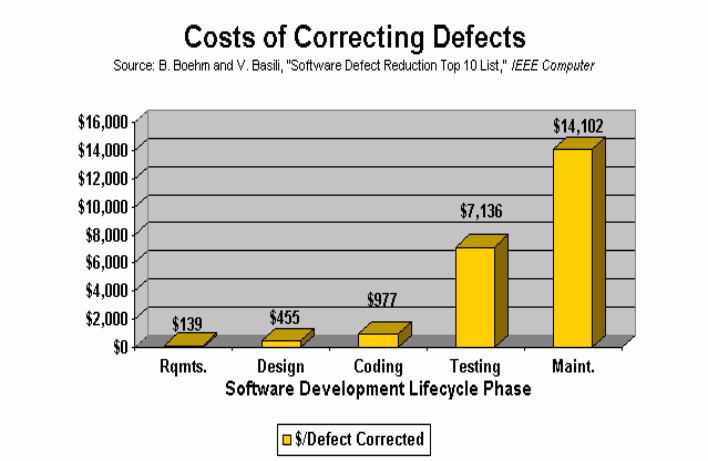

Fig. 11 Cost of Fixing a Defect [9]

The resulting ***cost savings of $2,276,700 represent an ROI of approximately 400%, even after engagement costs are taken into consideration***. Saving grow with the number of defects found (which is directly related to an application's size) and will also be realized during the application's maintenance phase, as new releases are developed and tested over time. Again, this ROI figure represents *only* money saved during the software development process and *does not* include money saved by avoiding the costs associated with reputation damage, lost productivity and liability costs should the software fail to function properly in the marketplace.

Other benefits:

- IOSTP solutions allowed the client to deliver a higher quality product in significantly less time.

- The client is able to realize enterprise-wide savings and efficiencies by integrating IOSTP methodology into its future software development efforts.

An important area to focus on when optimizing your testing is identifying all of the business and technical requirements that exist for an application, and then prioritizing them based on the impact of failure on the business. QA teams should ensure they have access to the application's business and technical requirements in order to create effective test requirements. Involving business managers, test managers and QA architects will help achieve the balance of testing that is optimal [3-8]. The advantage of automated risk-based testing is that it adds a level of objectivity not available with traditional testing, where individual testers were left to determine what should be tested and when. Thoroughly understanding and correctly prioritizing testing requirements can have the greatest impact on successful delivery of a high-quality application. By implementing an optimized testing solution, IT can ensure quality activities accurately reflect business priorities, and can make certain they are testing the right areas of an application within the constraints of the schedule. Using an optimized testing solution, the risks are calculated automatically and time estimates are rolled up per requirements balancing quality, schedule and cost through risk-based practices. This allows testers to apply a time factor to existing risk factors, which enables users to quickly select the highest-priority test cases and understand how long it will take to test them [4].

4 Software Testing Optimization Model and IT benefits

With optimized testing, IT organizations are able to balance the quality of their applications with existing testing schedules and the costs associated with different testing scenarios. Optimized testing provides a sound and proven approach that allows IT to align testing activities with business value. The practices, processes and tools that encompass optimized testing offer many benefits. The increasing cost and complexity of software development is leading software organizations in the industry to search for new ways through process methodology and tools for improving the quality of the software they develop and deliver.

4.1 Manage "what-if" scenarios -A Software Testing Optimization Model

Such scenarios are invaluable for determining where testing resources should be spent at the beginning of software development project. With an optimized testing solution, you can create what-if scenarios to help users understand the impact of changing risks, cycle attributes and requirements as priorities change. This insight proves invaluable when a testing organization is trying to determine the best way to balance quality with cost and schedule. By understanding the impact of different factors on testing, IT managers can identify the right balance.

We applied the End-to-End (E2E) Test strategy in our Integrated and Optimized Software Testing framework (IOSTP) [3-5]. End-to-End Architecture Testing is essentially a "gray box" approach to testing - a combination of the strengths of white box and black box testing. In determining the best source of data to support analyses, IOSTP with embedded RBOSTP considers credibility and cost of each test scenario i.e. concept. Resources for simulations and software test events are weighed against desired confidence levels and the limitations of both the resources and the analysis methods. The program manager works with the test engineers to use IOSTP with embedded RBOSTP [4] to develop a comprehensive evaluation strategy that uses data from the most cost-effective sources; this may be a combination of archived, simulation, and software test event data, each one contributing to addressing the issues for which it is best suited.

The central elements of IOSTP with embedded RBOSTP are: the acquisition of information that is credible; avoiding duplication throughout the life cycle; and the reuse of data, tools, and information. The system/software under test is described by objectives, parameters i.e. factors (business requirements - BR are indexed by j) in requirement specification matrix, where the major capabilities of subsystems being tested are documented and represent an independent i.e. input variable to optimization model. Information is sought under a number of test conditions or scenarios. Information may be gathered through feasible series of experiments (E): software test method, field test, through simulation, or through a combination, which represent test scenario indexed by i i.e. sequence of test events. Objectives or parameters may vary in importance α_j or severity of defect impacts. Each M&S or test option may have k models/tests called modes, at different level of credibility or probability to detect failure β_{ijk} and provide a different level of computed test event information benefit B_{ijkl} of experimental option for cell (i,j), mode k, and indexed option l for each

feasible experiment depending on the nature of the method and structure of the test. Test event benefit B_{ijkl} of feasible experiment can be simple ROI or design parameter solution or both etc. The cost C_{ijkl}, of each experimental option corresponding to (i,j,k,l) combination must be estimated through standard cost analysis techniques and models. For every feasible experiment option, tester should estimate time duration T_{jikl} of experiment preparation end execution. The testers of each event, through historical experience and statistical calculations define the E_{ijkl}'s (binary variable 0 or 1) that identify options. The following objective function is structured to maximize benefits and investment in the most important test parameters and in the most credible options. The model maintains a budget, schedule and meets certain selection requirements and restrictions to provide feasible answers through maximization of benefit index - $B_{enefit}I_{ndex}$:

$$B_{enefit}I_{ndex} = \max_{i,j,k,l} \sum_{j} \sum_{i} \sum_{k} \sum_{l} \alpha_j \beta_{ijk} B_{ijkl} E_{ijkl} \quad (4)$$

Subject to:

$$\sum_{j}\sum_{i}\sum_{k}\sum_{l} C_{ijkl}E_{ijkl} \le BUDGET$$ (Budget constraint);

$$\sum_{j}\sum_{i}\sum_{k}\sum_{l} T_{ijkl}E_{ijkl} \le TIMESCHEDULE$$ (Time-schedule constraint)

$\sum_{l} E_{ijkl} \le 1$ for all i,j,k (at most one option selected per cell i, j, k mode)

$\sum_{k}\sum_{l} E_{ijkl} \ge 1$ for all i,j (at least one experiment option per cell i, j)

4.1.2 Defect metrics as a RBOST drivers

A defect is defined as an instance where the product does not meet a specified characteristic. The finding and correcting of defects is a normal part of the software development process. Defects should be tracked formally at each project phase. Data should be collected on effectiveness of methods used to discover defects and to correct the defects. Through defect tracking, an organization can estimate the number and severity of software defects and then focus their resources (staffing, tools, test labs and facilities), release, and decision-making appropriately. Two metrics provide a top-level summary of defect-related progress and potential problems for a project: -defect profile and defect age. The defect profile chart provides a quick summary of the time in the development cycle when the defects were found and the number of defects still open. It is a cumulative graph. The defect age chart provides summary information regarding the defects identified and the average time to fix defects throughout a project. The metric is a snapshot rather than a rate chart reported on a frequent basis. The metric evaluates the "rolling wave" phenomenon, where a project defers difficult problems while correcting easier problems. In addition, this measure provides a top-level summary of the ability of the organization to successfully resolve identified defects in an efficient and predictable manner. If this metric indicates that problems are accumulating in the longer time periods, a follow-up investigation should be initiated to determine the cause.The metric evaluates the rolling wave risk where a project defers difficult problems while correcting easier or less complex problems [4]. In addition this measure will indicate the ability of the organization to successfully resolve identified defects in an efficient and predictable manner. If this metric indicates that problems are taking longer than expected to cl ose the schedule and cost risks increase in likelihood and a problem may be indicated in the process used to correct problems and in potentially in the resources assigned.

In next section we describe analytical optimization model of IOSTP process [3].

4.1.3 Defect removal efficiency model

When detected through walkthroughs, peer reviews inspections or testing, defects should be corrected effectively, requiring only one re inspection or regression test to verify removal as shown in Fig.12. If the software test managers require more than one iteration through the defect removal process, then those processes may require improvement.

The defect removal effectiveness metric tracks the history of these defect removals. For demonstration purpose we identified these SDLC phases denoted by **P**: Requirement (**P=1**), HL Design (Architecture level – **P=2**), LL Design (Detailed design – **P=3**), Code (Unit) test (**P=4**), Integration/System Test (**P=5**), Acceptance (User) Test (**P=6**), and Operation (Maintenance – **P=7**). For P=1 i.e. Requirement phase it is obvious that $D_{InP}=0$ and that $D_{InP}= D_{LP-1}$ for the rest P. If D_{dP} represent total defect detected in phase P, then $D_{dfP} \le D_{dP} \le D_{TP}$, because of defect fixing priority i.e. some of

detected defect in P are defered (postponed) to fix later. From our expe rience, rework calculated as percent of defect fixes returned $n_{average}$=3 times (regression test cycles) to development is in Average=10.5%, Std_Dev=6.6%. Finaly $DD_P = \frac{D_{dP}}{D_{TP}}$ denotes Defect Detection rate in phase P. Some representative **Defect Removal Efficiency and defect fixing Cost** matrix data that we call **DRECR** of system/software under test described by objectives, parameters i.e. factors (indexed by *j*) in requirement specification matrix from few project versions history is presented in Table 7.
If a large number of fixes are ineffective, then the process used for corrections should be analyzed and corrected. Items to report include:

1. Total inspections to be conducted or tests to run
2. Inspections or tests completed
3. Cumulative inspections or tests failed

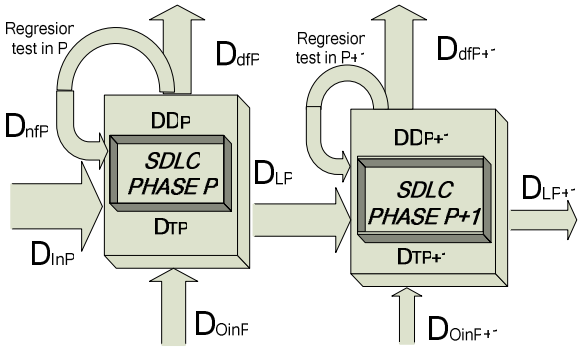

Fig. 12 Defect Removal Efficiency Model where: D_{InP} - denotes defects escaped from previous SDL phase P, D_{OinP} - denotes defects originated (introduced) in phase P, D_{TP} – denotes total existed defects in phase P, D_{dfP} - denotes defects fixed in phase P, D_{nfP} - denotes defects fixed in phase P after n regressions cycles, D_{LP} denotes defects leakage in phase P (escaped to phase P+1),DD_P - denotes Defects Detection rate in phase P.

The final test metric relates to technical performance testing. The issues in this area vary by type of software being developed, but top-level metrics should be collected and displayed related to performance for any medium- or high- technical risk areas in the development. The maximum rework rate was in the requirements which were not inspected and which were the most subject to interpretation. Resolution of the defects and after the fact inspections reduced the rework dramatically because of **Defect Containment.** Defect containment metric tracks the persistence of software defects through the life cycle. It measures the effectiveness of development and verification activities. Defects that survive across multiple life-cycle phases suggest the need to improve the processes applied during those phases.

4.1.4 Cost to fix error

For each development phase, the number of defects detected during that phase shall be tracked. In addition, for each defect, the phase in which that defect was created shall be tracked. If defects from earlier phases are not detected during that phase, there may be a need to improve the processes used for exiting those phases. Such defects suggest that additional defects are latent as presented in Table 1. The last column represent relative *Additional Cost to Repair Multiplier* ratio range $CM_{s=1\ j\ P\to P+1} - CM_{s=5\ j\ P\to P+1}$ for errors with lowest severity ***s=1 and highest severity 5*** of error originated in previous *P* phase but escaped and detected in later *P+1* phase compared to cost to fix immediately using cost to fix of Requirement defect as a base i.e. 1.

4.1.5 Defect age leading indicator

The Defect Age Metric will summarize the average time to fix defects. The purpose of this metric is to determine the efficiency of the defect removal process and, more importantly, the risk, difficulty and focus on correcting difficult defects in a timely fashion. The metric is a snapshot rather than rate chart reported on a frequent basis.
The metric evaluates the rolling wave risk where a project defers difficult problems while correcting easier or less complex problems. In addition this measure will indicate the ability of the organization to successfully resolve identified defects in an efficient and predictable manner. If this metric indicates that problems are taking longer than expected to close the schedule and cost risks increase in likelihood and a problem may be indicated in the process used to correct problems and in potentially in the resources assigned.

4.1.6 Risk summaries and reserve

Effective continuous risk management requires risk visibility. The best top-level indicators for summary risk management are the risk summary and reserve charts. Cost and schedule risk reserves should be established at the beginning of the project to deal with unforeseen problems. Risk summary and reserve charts show the total risk

exposure for cost and schedule compared with the current cost and time risk reserves for the project. The cost and time risk reserve for a project will change over time as some of this reserve is used to mitigate the effects of risks that actually occur and affect the project. Some Risk Management Performance (RMP) metrics are defined in [3-5] such are: Risk Growth Performance Index (RGPI), Risk Cost Performance Index (RCPI) etc. The charts that show both the total identified risk values and the probabilistic weightings of occurrence are very useful. As risks are actualized without complete abatement, or resources are expended in the risk-abatement process, the risk reserves are adjusted downward accordingly. An example display of a cost risk summary and reserve chart is provided below. Schedule risk summary and reserve charts are similar, but reflect schedule risk instead of cost risk.

5 Economic value measurement as leading indicators of optimization

For simplicity purpose, an undetected major or higher severity (***s≥4, s=1..5***) defect that escapes detection and leaks to the next phase may cost ten times to detect and correct. A minor or lower severity (***s≤3***) defect may cost two to three times to detect and correct. The Net Savings (NS) then are nine times for major defects and one to two times for minor defects. Because of that we apply simple but proven reasoning aout high ROI as key benefit of software test events B_{ijkl} in optimization objective equation (1) i.e. ***ROI_j= Net Savings for j objective/Detection Cost for j objective***. Of course, some benefits of the system/software under test described by objectives, parameters i.e. factors (indexed by ***j***) in requirement specification matrix, which is the major capabilities of subsystems being tested, must be verified and validated in every SDLC phase P by many test events. Of course, few objectives are tested only in one or two phases P and test events. Also, *Net Savings for j objective in phase P: Cost Avoidance-Cost to detect/Repair Now in phase P*.

It means, *Net Saving* benefit is error prevention to escape from phase P to next P+1 phase, or downstream phases to the customer use of defective software in the field. In mathematics language, it is calculated as:

$$NS_{ijkl} = \sum_{P=1}^{7} \delta_{j\,P} * p_{ijkl\,P} * CA_{ijkl\,P} \qquad (5)$$

where $\delta_{jP} = 0$ *if not aplicable in phase P,* 1 *if is applicable in phase P*, $p_{ijkl\,P}$ is probability of feasible l of k experiments in phase P to detect error of *j objective i.e to prevent defect to escape in phase P+1*. Also, $\sum_{P=1}^{7} \delta_{j\,P} p_{ijkl\,P} = 1$, and cost avoidance $CA_{ijkl\,P}$ in phase P is calculated as:

$$CA_{ijkl\,P} = \sum_{r=1}^{P} DD_{s\,j\,r\to P} * CM_{s\,j\,P\to P+1}, \text{ or}$$

rewritten as,

$$CA_{ijkl\,P} = \sum_{r=1}^{P} DD_{s\,j\,r\to P} * (CM_{s\,j\,P+1} - CM_{s\,j\,P}) \qquad (5)$$

DEFECTS ⇩ / FOUND IN: ⇨		Requirement	HL Design	LL Design	Code (unit) Test	Integration/ System Test	Acceptance (User) Test	Operation Post-Release	**Total orig. in phase**	**Fixing multiplier (Cost Ratio) CM_1 - CM_8**
ORIGINATED IN:	Requirement	22	3	8	2	4	4	2	45	1
	HL Design	0	17	9	1	2	2	3	34	3-6
	LL Design	0	0	12	11	8	5	4	40	8-10
	Code (unit) Test	0	0	0	7	9	8	1	25	10-15
	Integration/ System Test	0	0	0	0	4	2	2	8	15-40
	Acceptance (User) Test				0	0	2	1	3	30-70
	Operation Post-Release	0	0	0	0	0	0	0	0	40-1000
	Total found in phase	22	20	29	21	27	23	13	155	

Table 7 Typical Defect Removal Efficiency and defect fixing Cost Ratio matrix DRECR

where $DD_{s\ j\ r\to P} = DRECR(r,P)$ denotes Defect Detected in phase P of j objectivity, s severity for defects $D_{sj\ OrIn\ r}$ originated in phase r but escaped and detected in phase P denoted as $D_{sj\ dOrIn\ r\to P}$ that will make additional cost to detect and fix by cost multiplier $CM_{sj\ P\to P+1}$. Cost avoidance in phase P, then will be easily calculated from DRECR matrix like

$$CA_{ijkl\ P} = \sum_{r=1}^{P} DRECR(r,P) * (CM(P+1) - CM(P))$$

Finaly, if j *objective* severity ($s=1..5$) is assesed in requirement or specification matrix than importance $\alpha_j=s$, $\beta_{ijk} = p_{ijkl\ P}$ of experiment i.e. we must offer as many as we could feasible k series of experiments (E): software test method, field test, through simulation, or through a combination, which represent test scenario indexed by ***i*** to find out maximal benefit index -$B_{enefit}I_{ndex}$ rewritten as:

$$B_{enefit}I_{ndex} = \max_{i,j,k,l} \sum_{j} \sum_{i} \sum_{k} \sum_{l} s_j ROI_{ijkl} E_{ijkl} \quad (6)$$

Where, $ROI_{ijkl} = \dfrac{NS_{ijkl}}{C_{ijkl}}$ and (budget, cost) constraints as in (4).

This model goal is to find out test scenario indexed by ***i*** with maximal benefit index -$B_{enefit}I_{ndex}$ based on Return on Investment bases and appropriate Risk Management activities assure the savings on the cost avoidance associated with detecting and correcting defects earlier rather than later in the product evolution cycle.

6 Conclusion

Although it is important to measure the quality of the product under development, it is equally important to measure the effectiveness and efficiency of Software Testing itself as an activity – not a service. We proposed basic metrics of key software testing activities and artifacts in development processes that can be objectively measured, according to ISO 15939 – Software Measurement as a foundation for enterprise wide improvement of Integrated and Optimized Software Development / Testing Pocess (IOSTP) [3-5] i.e. **S**oftware **T**esting **M**etrics **F**ramework (STMF). Specifically, the measurements described in this paper first answers the question of whether Software Testing is "doing the right thing" (effectiveness).

Once there is assurance and quantification of correct testing, metrics should be developed that determine whether or not Software Testing "does the thing right" (efficiency). By measuring effectiveness and efficiency, a Software Testing organization can better communicate its own importance using factual information.

References:

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

[2] V. R. Basili, G. Caldiera, H. D. Rombach, The Goal Question Metric Approach, Encyclopedia of Software Engineering, volume 1, John Wiley & Sons, 1994, pp. 528-532

[3] Lj. Lazić, The Integrated and Optimized Software Testing Process, *PhD Thesis, School of Electrical Eng., Belgrade, Serbia,* 2007.

[4] Lj. Lazić, Mastorakis, N. RBOSTP: Risk-based optimization of software testing process Part 2", *WSEAS TRANSACTIONS on INFORMATION SCIENCE and APPLICATIONS, Issue 7, Volume 2, p 902-916,* July 2005.

[5] Lj. Lazić, N. Mastorakis, A Framework of Software Testing Metrics – Part 2, 11h WSEAS CSCC'08 Multiconference, Agios Nikolaos, Crete Island, Greece, July 23-28, 2007., 200X.

[6] J. Capers, *Estimating Software Costs*. 2nd edition. McGraw-Hill, New York: 2007

[7] J. Capers, *Applied Software Measurement*. 3rd edition; McGraw-Hill, New York: 2008..

[8] G. Tassey, The Economic Impacts of Inadequate Infrastructure for Software Testing. National Institute of Standards and Technology,2002, <www.nist.gov/director/ progofc/report02-3.pdf>.

[9] B. Boehm and V. Basili, Software Defect Reduction Top 10 List, *IEEE Computer*, IEEE Computer Society, Vol. 34, No. 1, January 2001, pp. 135-137.

[10] Bennett, Ted L., and Paul W. Wennberg. "Eliminating Embedded Software Defects Prior to Integration Test."Dec. 2005.