A Framework of Software Testing Metrics – Part 2

Ljubomir Lazić, School of Electrical Engineering, Vojvode Stepe 283, Beograd, SCG
Nikos Mastorakis, Military Institutions of University Education, Hellenic Naval Academy, Terma Hatzikyriakou, 18539, Piraeus, Greece

Abstract:- Software testing needs to be measured in similar terms as overall software development process (SDP) in order to understand its true progress and make informed decisions. Basic considerations of **S**oftware **T**esting **M**etrics **F**ramework (STMF) and some commonly used testing metrics and where in testing process they apply are described in this, Part 1 article. Typically, software development is measured in terms of overall progress in meeting functional and business goals. By considering testing dimensions other than cost and schedule, managers and other team members can better understand and optimize the testing process, in effect opening the black box and managing testing more effectively were described in Part 2 article.
Key-Words:- software testing, testing metrics, size estimation, effort estimation, test effectiveness evaluation.

1 Introduction

By considering testing dimensions other than cost and schedule, managers and other team members can better understand and optimize the testing process, in effect opening the black box and managing testing more effectively. In this way they can avoid costly and painful "surprises" late in the project.

Test metrics are an important barometer used to measure the effectiveness of the software testing process. In our Part 1 article [1], the basic considerations of **S**oftware **T**esting **M**etrics **F**ramework (STMF) and some commonly used testing metrics and where in testing process they apply are described. Aim of this Part 2 article is to explain in more detail proposed basic metrics of key software testing activities and artifacts in development processes that can be objectively measured, according to ISO 15939 – Software Measurement and SEI CMMI-SE/SW/IPPD/SS product suit [2-4] as a foundation for enterprise wide improvement of Integrated and Optimized Software Development / Testing Process (IOSD/IOSTP) [9-13] i.e. **S**oftware **T**esting **M**etrics **F**ramework (STMF).

Improvements in the software development process depend on our ability to collect and analyze data drawn from various phases of the development life cycle. Our design metrics research team was presented with a large scale SDP production model plus the accompanying problem reports that began in the requirements phase of development. The goal of this research was to identify and measure the occurrences of faults and the efficiency of their removal by development phase in order to target software development process improvement strategies. Through our analysis of the system data, the study confirms that catching faults in the phase of origin is an important goal. The faults that migrated to future phases are on average ten times more costly to repair. The study also confirms that upstream faults are the most critical faults and more importantly it identifies detailed design as the major contributor of faults, including critical faults.

In testing we tend to focus on collecting internal IOSTP measures such as numbers of defects and innovation measures such as process improvement metrics. If we examine where our normal test metrics fit in the STMF we can see gaps in both quantitative and qualitative measures, which we may wish to address not only to focus on internal IOSTP results but to look at a balance between four measurable areas: financial measures such as profit and loss, customer measures such as market share and repeat business, internal measures such as numbers of defects in products and process violations, and innovation or learning measures such as number of new products developed and marketed. To achieve useful accuracy, software quality models must be calibrated for each specific development environment [2]. A case study acquires historical data on one or more projects. We construct models that could have been developed during the historical project, and calculates assessments that could have been made. The accuracy of those assessments is then evaluated against actual experience. This gives us confidence in predictions for a current project. Exploit your gold mines. Our approach to software quality modeling is aptly described as data mining, especially when operational faults are rare and vary significantly from one to another source of information. Data mining is most appropriate when one seeks valuable bits of knowledge in large amounts of data collected for some other purpose, and when the amount of data is so large that manual analysis is not possible. Many software development organizations have very large databases for project management, configuration management, and problem reporting which capture data on individual events during development. For large systems or product lines, the amount of available data can be overwhelming. Manual analysis is certainly not possible. However, we have found that these databases do contain indicators of which modules will likely have operational faults [12].

One metric is not enough. Much of the literature on software metrics is aimed to demonstrate the value of

individual metrics. However, this does not fulfill our purpose: to build industrial-strength quality models.
Our experience with modeling empirical data from industry has indicated that a model with one software metric as the only independent variable does not have useful accuracy and robustness. Lines of code is not enough. McCabe cyclomatic complexity is not enough. The metric that is most highly correlated to faults is not enough. Recent case studies have demonstrated that multiple independent variables give more accurate results than models with just one independent variable [5]. The cost of collecting many metrics from source code (or other software product), rather than just a few, is not a practical problem for conventional metrics, because a metric-analyzer software tool is capable of measuring many metrics in one pass. We have found it is more effective to begin with many metrics, and then to apply data mining techniques to choose those with statistically significant empirical relationships to faults. Code metrics are not enough. The development histories of modules often differ radically. For example, modules from early releases have been used or tested more than recently developed modules. A stable module may have been developed by only one person, while other modules may have been modified by many different programmers. Indicators of such variations can significantly improve model accuracy and robustness. For example, our case studies have shown that a simple indicator of reuse from a prior release can be a significant independent variable in both classification and regression models. A case study of the Automatic Target Tracking Radar System - ATTRS [10], showed that the likelihood of discovering additional faults during integration and test in a spiral life cycle can be usefully modeled as a function of the module history prior to integration. In other words, process-related measures derived from configuration management data and problem reporting data may be adequate for software quality modeling, without resorting to software product measurement tools and expertise. Empirical validation must be realistic. Due to the many human factors that influence software reliability, controlled experiments to evaluate the usefulness of empirical models are not practical. Therefore, we take the case study approach to demonstrate their usefulness in a real-world testing. To be credible, the software engineering community demands that the subject of an empirical study be a system with the following characteristics [6]: (1) developed by a group, rather than an individual; (2) developed by professionals, rather than students; (3) developed in an industrial environment, rather than an artificial setting; and (4) large enough to be comparable to real industry projects. Our case studies fulfill all of these criteria through collaborative arrangements with development organizations. The analysis presented here has study data that is especially useful since the data supplied was compiled as early as the requirements phase. Such thorough fault reporting is relatively uncommon and is most helpful in determining the origin and resolution of faults in the development process.

In section 2, the SW Testing Measurement Infrastructure and some commonly used testing metrics and where in testing process they apply are described. Software Testing Metrics Framework deployment in our IOSTP as a case study is described in section 3. Finally in section 4, some concluding remarks are given.

2 The SW Testing Measurement Infrastructure

You can't track project status meaningfully unless you know the actual effort and time spent on each task compared to your plans. You can't sensibly decide whether your product is stable enough to ship unless you're tracking the rates at which your team is finding and fixing defects. You can't quantify how well your new development processes are working without some measure of your current performance and a baseline to compare against. Metrics help you better control your software projects and learn more about the way your organization works trough Metrics Life Cycle as depicted in figure 1. 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 [11,12]. 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. The analysis presented here has study data that is especially useful since the data supplied was compiled as early as the requirements phase. Such thorough fault reporting is relatively uncommon and is most helpful in determining the origin and resolution of faults in the development process. As we described in our Part 1 article, the Goal Question Metric (GQM) process, created by Victor Basili and his colleagues at the University of Maryland, an excellent technique for selecting appropriate metrics to meet the specific measurement needs of an organization [8,9], see figure 2. The data consisted of the IOSD/IOSTP production model and the related problem reports for the model. Our research team had the task of tracking each fault identified in the problem reports back to its software component. Each problem report consisted of 35 fields that included the development cycle phase of origin and phase found, severity class, a fault class, detection method and the amount of effort required to resolve the fault. In addition, a separate analysis section was appended to

each report detailing the description of the problem, the problem history, the suggested cause and solution and subsequent changes to the model.

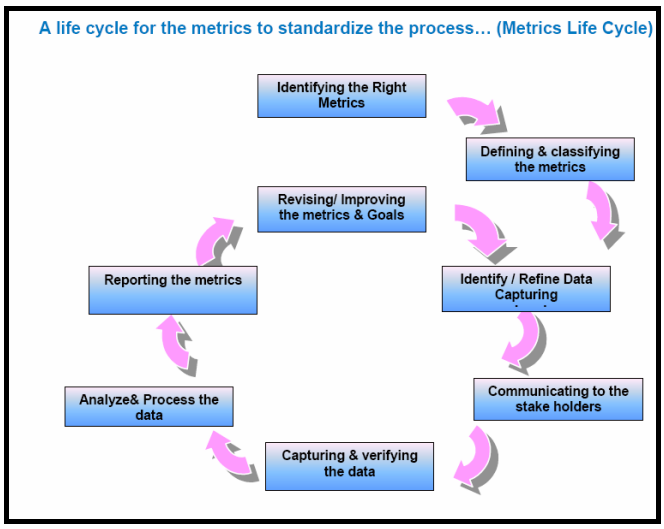

Figure 1. Metrics life cycle in STMF

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.

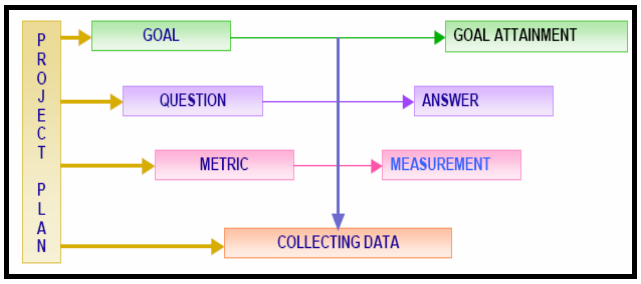

Fig 2. Goal Question Metric (GQM) process

This enables Software Testing organizations to break free from the misconception that Software Testing measurement should concentrate on issues important to the Software Development community. Often, there are early warning signs that testing is going to have problems.

In summary form [2], successful software projects in the 10,000 function point class usually are characterized by these attributes:

1. Less than 1% monthly requirements changes after the requirements phase.
2. Less than 5.0 defects per function point in total volume.
3. More than 65% defect removal efficiency before testing begins.
4. More than 94% defect removal efficiency before delivery.

In summary form, unsuccessful software projects in the 10,000 function point class
usually are characterized by these attributes:

1. More than 2% monthly requirements change after the requirements phase.
2. More than 6.0 defects per function point in total volume.
3. Less than 35% defect removal efficiency before testing begins.
4. Less than 85% defect removal efficiency before delivery.

These show up in the details of the analysis and design phases of the tests themselves. They appear in the form of incomplete or deferred work due to missing information, improperly managed problems recorded against key functionality, and other "small" indicators accumulating over time. If these indicators are spotted far enough ahead of time by managers, developers, and the testers themselves, work can be done to head problems off while they are still small. This in turn ensures that the testing group is better prepared for the software and that the software is better prepared for testing. Organizations can avoid last-minute quality issues by addressing testing problems earlier in the process, when they are still small. Doing this requires better insight into a project than what can be gotten from a Gantt chart. During test development, management needs to know the status of test planning and preparation to properly gauge the readiness of the test team to test the software. To support these different needs, different levels of detail in each of the following categories must be provided to each group.

1. *Schedule:* What tests will be run? When will the tests be ready? How much effort will it take? When will it be complete?

2. Functionality: What requirements will be tested and where? How will tests divide up application requirements? How much of the functionality has been tested for a given version of the software?

3. *Code:* What parts of the code are exercised by the tests? What problems have been found? How much of the code in a given version has been executed during testing?

4. *Problems:* What problems are tested for? What problems have been found? How significant are the problems? What parts of the software are affected by the problems? What versions are affected by the problems? What requirements are impacted by these problems? What is the impact of these problems on the testing?

The questions posed for each of these areas must be carefully examined in order to properly understand and track the status of project test activities. In addition, having a solid understanding of the planning and preparation requirements for each testing phase is

critical to making correct decisions about project schedule, status, and release.

Table 1 below summarizes commonly used testing metrics and where in the testing process they apply.

Metric Type	Test Development Metrics	Test Execution Metrics
Functional Metric	• Number of requirements allocated by test • % of requirements by test development phase	• Number of requirements verified • % of requirements tested by version • % of requirements tested by major software component • Stability of server/platform per user
Code Metric	• % of code covered per test • % of code coverage per major software component	• Code coverage of tests completed for each version under test
Problem Metrics	• Problems tested for in regression tests • Extreme conditions tested for in functional tests	• Problems found per version tested • Problems found per software component • Number of critical/high problems found per version
Schedule Metrics	• % completion of functional test requirements by testing phase • Weighted functional requirement completion	• Tests completed per version • Estimated number of days to complete • Test cycle completion time • Time to complete testing per functional area

Table 1. Commonly Used Testing Metrics

You can't start with all of these, but we recommend including at least the following measurements early in your metrics program:

• *Product size*: count lines of code, function points, object classes, number of requirements, or GUI elements

• *Estimated and actual duration* (calendar time) and *effort* (labor hours): track for individual tasks, project milestones, and overall product development

• *Work effort distribution*: record the time spent in development activities (project management, requirements specification, design, coding, testing) and maintenance activities (adaptive, perfective, corrective)

• *Defects:* count the number found by testing and by customers and their type, severity, and status (open or closed)

3 Measuring the Test Process: IOSTP case study

In this section we describe some test metrics as contribution to Risk-Based Optimization of Software Testing Process i.e. RBOSTP [11]. which is designed to improve the efficiency and effectiveness of the testing effort by combining Earned (Economic) Value (EV), Risk Management (RM) strategy. Based on a proven and documented Integrated and Optimized Software Testing methodology (IOSTP) [8-12]. The IOSTP with embedded RBOSTP help organizations reduce project risk and significantly lower the cost of defects. It focus on solving the problems of delivering high quality software on time and at an affordable price with simulation-based software testing scenarios to manage stable (controllable and predictable) software testing process at lowest risk.

3.1 E2E Test Concepts – Optimized test scenario

Traditional approaches, such as module and integration testing, only addressed part of the need. Module testing checks individual units and integration testing checks subsets of modules. But neither addresses the quality of the overall system, particularly in systems of systems. Neither test approach addresses the quality of the system from the end user's point of view. What is End-to-End Testing & Assurance-Based Testing [12], shortly *E2E testing* and how does it address those concerns? E2E testing focuses on the end user's point of view. As we know the entire system work together to produce the correct, desired end result for the user. It documents paths that can be traced through the modules and subsystems to produce an output or function that serves the user correctly (see figure 3). Once these paths are identified, they can be ranked for risk and criticality. The ranking forms the basis for selecting test scenarios wisely. E2E represent the user's point of view using thin threads to define the user's perspective in E2E test specifications. A thin thread represents a minimum usage scenario of an integrated system. Essentially, a thin thread is a complete scenario from the end user's perspective; the system takes input data, performs some computation and produces output. The thin thread describes the whole scenario and it describes just one function. Thin threads with certain commonalities can form a hierarchical thin thread group. That is, a collection of low-level, thin thread groups with certain commonalities can be further grouped into a high-level, thin-thread group. In this way, all thin threads and thin

thread groups can be arranged into a thin-thread tree of a banking system as an example (see figure 4). The root of a thin thread tree represents the overall integrated system under test (SUT), a branch node represents a collection of related thin threads (thin thread group), and a leaf represents a concrete thin thread. Thus, a thin hread tree can be viewed as a functional decomposition of the system under test. A condition is a companion concept to the thin thread. Numerous conditions affect the execution of a thin thread. A thin thread is activated when all its affiliated conditions are satisfied. The possibilities include communication conditions, sequencing and timing conditions, data conditions, and environmental conditions. Like thin threads, conditions can be organized into a tree structure to facilitate reuse and management.

Thin threads and conditions are the core of E2E test specifications. They provide a very effective framework for risk analysis, test case generation, and regression testing. How does E2E support risk analysis? The test engineer begins by examining two factors: the probability that a thin thread testing will fail and the consequences of the failure if it occurs. The risk assigned to a thin thread is a function of its failure probability and the consequence of its failure. The level of risk can then also be calculated for each condition and test case. E2E approach provide easy and effective vay to generate test cases. An E2E test case is built on either a basic scenario (thin thread) or a complex scenario (combination of thin threads). In either case, it is defined by a set of input data and the expected outputs. A test case can be generated through these steps:

• Identify the subsystems involved, including both software and hardware.
• Identify the input data for the thread.
• Use the input data that satisfies the conditions associated with the thread.
• Determine the expected results fromthe thin-thread description.

Input data must be selected with care. Often, a thin thread is affected by several conditions, and each condition can be satisfied by multiple input data. In this case, a tester may need to exercise care in selecting proper test inputs.

Relationships among thin threads are useful in scheduling test case execution. For example, if a thin thread is on a critical path, it should be tested as early and asthoroughly as possible. If a set of thin threads will be selected for testing, it maybe appropriate to select thin threads with independent execution paths to ensure certain kinds of coverage. E2E approach support regression testing ensuring adequate regression testing with ripple effect analysis (REA) to analyze and eliminate the side effects of software changes and to ensure consistency and integrity after changes are made. REA is an iterative process of change request, software modification, impact identification, and validation. E2E supports REA because its test specifications embody both trace ability and dependency information. E2E captures traceabilty information by linking test scenarios to requirements, implementation, and test cases.

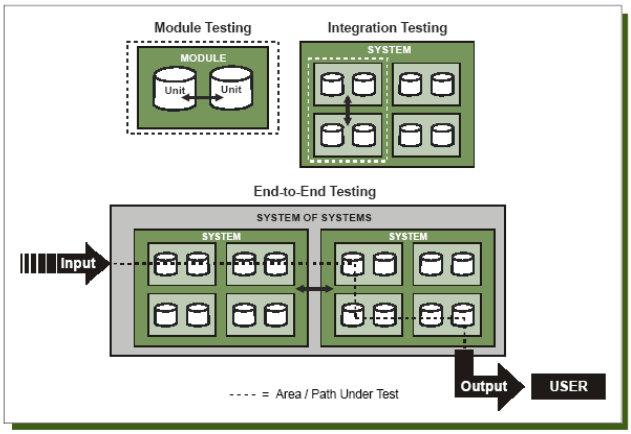

Fig. 3 E2E testing verifies that a system of systems will produce the correct
output from the end user's perspective.

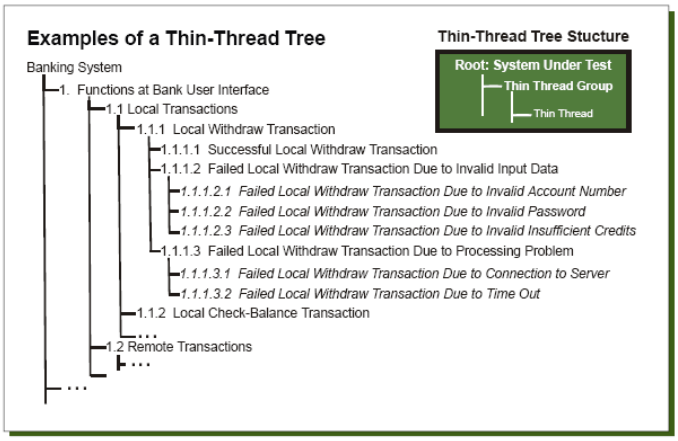

Fig. 4 Thin-thread trees provide a functional decomposition of the system
under test and form the foundation of E2E testing.

Traceability enables global change analysis among software artifacts. E2E links all the requirements associated with the test scenario and all the test cases generated from the test scenario, subsystem components, interfaces, and data. Dependency information allows an analyst to use test-slicing algorithms to detect scenarios that are affected by a change and thus are candidates for regression testing. There are few drawbacks or limitations to the E2E approach. E2E is meant for large, inter connected sub/systems. It can be very complex and requires an investment of time. But these systems demand an advanced level of effective testing, especially from the end user's perspective. E2E achieves that goal and more. E2E test specifications can be used as a functional specification for the system, as well as

training material for new engineers. This can be especially useful in large legacy applications, where many engineers have maintained the system over an extended period of time and no complete functional specification exists. Also, E2Etest specifications can be generated during system development and help derive design specifications in some formal or semi-formal notation. Also, E2E test specifications can serve as both the functional specification and test document for the project. This is useful when documents must be updated to reflect system changes. By using E2E test specifications for both purposes, fewer documents will need revision. This can save significant effort and resources. E2E is most effective when used in combination with **S**tatistical-**R**isk-**B**ased **T**est with **A**ssured **C**onfidence (SRBTAC) [12].

We applied the E2E Test strategy in our Integrated and Optimized Software Testing framework (IOSTP). 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 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 (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 -$B_{enefit}I_{ndex}$:

$$B_{enefit}I_{ndex} = \max_{i,j,k,l} \sum_{j} \sum_{i} \sum_{k} \sum_{l} \alpha_j \beta_{ijk} B_{ijkl} E_{ijkl} \qquad (1)$$

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)

Models and simulations can vary significantly in size and complexity and can be useful tools in several respects. They can be used to conduct predictive analyses for developing plans for test activities, for assisting test planners in anticipating problem areas, and for comparison of predictions to collected data. Validated models and simulations can also be used to examine test article and instrumentation configurations, scenario differences, conduct *what-if* tradeoffs and sensitivity analyses, and to extend test results. In other words, the software parameters are estimated on-line and the corresponding optimal actions are determined based on the estimates of these parameters. 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 specified what next testing policy should be employed and thus in turn what test suite or next test case should be generated in accordance with the new testing policy to maximize benefit index function (1) which is in nature uncertain i.e. includes risk to detect failure. Benefit index function depends of failure severity if the

chosen feasible experiments i.e. test event don't detect failure (α_j) because of test event capability i.e. probability to detect failure β_{ijk}. Because of risk impact and probability of success of test event, increasing cost and rework time to fix undetected SUT faults, we call our Optimization approach of Software Testing Process - Risk-Based. RBOSTP is based on defect removal (detection and fixing) metrics.

3.2 Defect metrics as a E2E concept 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, defect fixing effort and defect age (see figure 5).

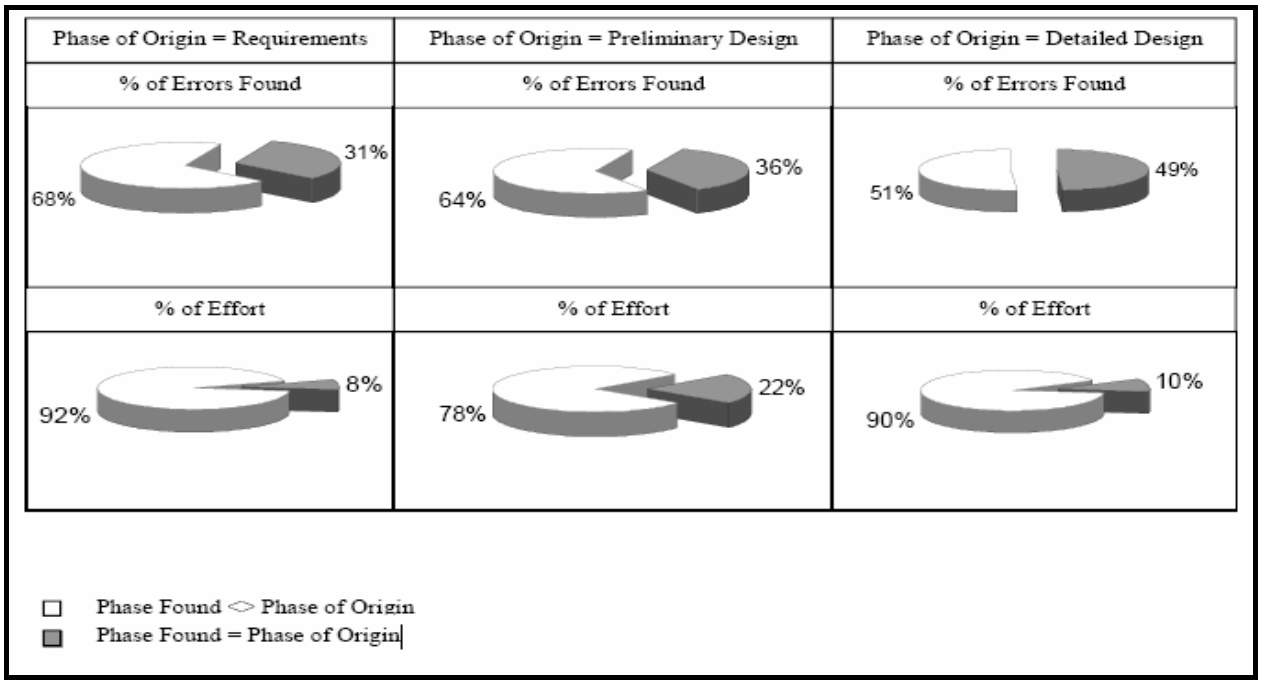

Fig. 5 Analysis of inter and intra-phase faults and removal effort by development phase

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 (see figure 6). 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.

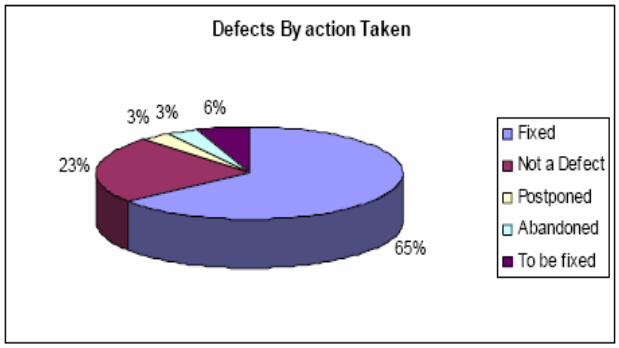

Fig. 6 Defect resolution metrics

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 (see figure 4 in our Part 1 [1] article, a follow-up investigation should be initiated to determine the cause. 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.

3.2.1 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. 7. 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 experience, 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.

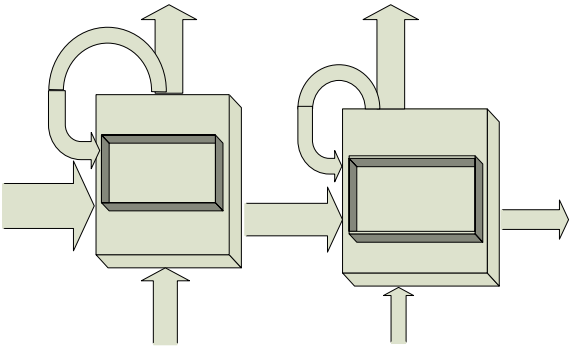

Fig. 7 Defect Removal Efficiency Model where: D_{InP} - denotes defects escaped from previous SDL phase P, D_{OinP} - denotes defects originated (introduced) in phase P, D_{TP} – denotes total existed defects in phase P, D_{dfP} - denotes defects fixed in phase P, D_{nfP} - denotes defects fixed in phase 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.

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 2.

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

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.

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 2 Typical Defect Removal Efficiency and defect fixing Cost Ratio matrix DRECR

3.2.2 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 $CM_{s=1\ j\ P\to P+1} - CM_{s=8\ j\ P\to P+1}$ for errors with lowest severity ***s=1 and highest severity 8*** 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.

3.3 Risk management and economic value measurement leading indicators 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 $\boldsymbol{B_{ijkl}}$ 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:

$$NS_{ijkl} = \sum_{P=1}^{7} \delta_{j\ P} * p_{ijkl\ P} * CA_{ijkl\ P} \qquad (2)$$

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:

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

$$CA_{ijkl\ P} = \sum_{r=1}^{P} DD_{s\ j\ r\to P} * (CM_{s\ j\ P+1} - CM_{s\ j\ P}) \qquad (3)$$

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 α_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:

$$B_{enefit}I_{ndex} = \max_{i,j,k,l} \sum_{j} \sum_{i} \sum_{k} \sum_{l} s_j ROI_{ijkl} E_{ijkl} \qquad (4)$$

Where, $ROI_{ijkl} = \frac{NS_{ijkl}}{C_{ijkl}}$ and (budget, cost) constraints as in (1).

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.

4 Conclusions

In software development organizations, increased complexity of product, shortened development cycles, and higher customer expectations of quality proves that software testing has become extremely important software engineering activity. Software development activities, in every phase, are error prone so defects play a crucial role in software development. At the beginning of software testing task we encounter the question: How to inspect the results of executing test and reveal failures? What is risk to finish project within budget, time and reach required software performance i.e. quality? How does one measure test effectiveness, efficacy, benefits, risks (confidence) of project success, availability of resources, budget, time allocated to STP?

How does one plan, estimate, predict, control, evaluate and choose "the best" test scenario among hundreds of possible (considered, available, feasible) number of test events (test cases)? Proposed Software Testing Metrics Framework as a part of IOSTP framework solved these issues combining few engineering and scientific areas such as: Software Measurement, 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.

References

[1]] Lj. Lazić, N. Mastorakis. A Framework of Software Testing Metrics – Part 2, 11h WSEAS CSCC (CIRCUITS-SYSTEMS-COMMUNICATIONS-COMPUTERS) Multiconference, Agios Nikolaos, Crete Island, Greece, July 23-28, 2007

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

[3] CMM Product Development Team. Capability Maturity Model for Software (CMM), Version 1.1, CMU/SEI-93-TR-24, ESC-TR-93-177. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA; 64 pp., 1993.

[4] CMMI Product Development Team. Capability Maturity Model Integration for Software Engineering (CMMi), Version 1.1, CMU/SEI-2002-TR-028, ESC-TR-2002-028. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA; 707 pp., 2002.

[5] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl. Data mining for predictions of software quality. *International Journal of Software Engineering and Knowledge Engineering*, 1999.

[6] L. G. Votta and A. A. Porter. Experimental software engineering: A report on the state of the art. In Proceedings of the Seventeenth International Conference on Software Engineering, *IEEE Computer Society*, pages 277-279, Seattle, WA, Apr. 1995.

[7] 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

[8] Lj. Lazić, N. Mastorakis. Software Testing Process Improvement to achieve a high ROI of 100:1, 6th WSEAS Int. Conf. On MATHEMATICS AND COMPUTERS IN BUSINESS AND ECONOMICS (MCBE'05), March 1-3, Buenos Aires, Argentina 2005.

[9] Lj. Lazić, D. Velašević, N. Mastorakis. A Framework of Integrated and Optimized Software Testing Process, WSEAS Conference, August 11-13, Crete, Greece, 2003 also in WSEAS TRANSACTIONS on COMPUTERS, Issue 1, Volume 2, January 2003.

[10] Lj. Lazić, D. Velašević. Applying Simulation and Design of Experiments to the Embedded Software Testing Process", SOFTWARE *TESTING, VERIFICATION AND RELIABILITY*, Volume 14, Number 4, p 257-282, John Willey & Sons, Ltd., 2004.

[11] 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, ISSN 1790-0832.

[12] Lj. Lazić, N. Mastorakis. "Faster, Cheaper Software Error Detection with Assured Confidence – Part 1", 5th WSEAS International Conference on APPLIED COMPUTER SCIENCE (ACOS '06), Hangzhou, China, Sponsored by WSEAS and WSEAS Transactions, Co-Organized by WSEAS and Zhejiang University of Technology, April 16-18, 2006.