A Large Software System Test Engineering Practices

Biswadeb Bandyopadhyay
Assistant Professor, Department of Computer Science and Engineering
University of Engineering and Management, Kolkata
E-Mail : biswadeb.bandyopadhyay@uem.edu.in
Mobile: +91-9903052768

1. Abstract

The following Paper describes the experiences of a test engineering team, which had worked with a large software product development and support activity. This team has studied the existing software product, available test tools, test environment, with an objective of analyzing existing testing processes and methodologies for this large software product. The Paper discusses a number of initiatives and recommendations made by this test engineering group aimed at increasing the testing efficiency, optimizing the test suites, measuring and improving effectiveness of test cases and the quantifiable benefits and process improvements, that can derived from such initiatives. This activity was undertaken as part of a test engineering initiative to bring in place a set of innovative test engineering practices as potential business value drivers.

2. Introduction

Software testing is a critical component in the software development life cycle**.** This begins right at the time the software development activity is started, and it continues in parallel with each phase of the development life cycle. An effective test approach, test strategy and test methodology will not only contribute towards improved product quality, but will also provide benefits in terms of reduction in software development cost, faster time to market and better acceptability of the product by the end user.

This Paper discusses the experiences and lessons learnt by a team of testing practitioners dealing with a large collection of test suites used for large software system maintenance. These test suites have evolved over the years and have been deployed to identify any regressions caused due to code enhancements or bug fixing in the code. In addition, they have been traditionally used for testing each release of the product. These test suites have been deployed to maintain the software system in terms of its reliability, serviceability and maintainability over the years.

Even though these test suites were found very effective, and have been contributing immensely to maintain the product quality over the years, a need was felt to make qualitative and quantitative improvements in these test suites by automating, optimizing and enhancing the test suites as part of continuous testing process improvement initiatives.

This Paper talks primarily about the challenges involved in and lessons learnt from

- Automating the execution of large collection of test suites
- Providing an integrated test environment to perform parallel testing, thereby bringing down the test execution time
- Optimizing the test suites
- Enhancing the test suites
- Developing a Web based code coverage analysis tool to generate code coverage statistics
- Developing a framework for using the code coverage analysis in developing new test cases
- Providing a Web based test environment to perform unit testing and code coverage analysis
- Innovating test engineering practices in testing on different platforms, product release testing, bug fix testing, developing test tools for a large software system

3. Background

Purpose of this Paper is to describe the initiatives taken by this group of software testing practitioners with an objective of analyzing existing testing processes and methodologies for this large software product. It discusses a number of initiatives and recommendations made by this test engineering team aimed at increasing the testing efficiency, optimizing the test suites, measuring and improving effectiveness of test cases and the

quantifiable benefits and process improvements, that can be derived from such initiatives.

Scope of this activity was to study the existing software product, available test tools, test environment and to make a number of recommendations to improve the testing processes, test tools and the test environment. The aim was to bring in place a set of innovative test engineering practices as potential business value drivers.

4. Approach and Detailed Description

Optimizing Test Cycle Time

The large software product that is being talked about has a large collection of test suites, which have evolved over a period of time. These test suites have been deployed over the years to identify any regression caused due to code enhancement, bug fixing in the code and used for testing each release of the product. These test suites have been found very useful in detecting large number of regressions and performing release testing of the product.

Size of such test suites was very large consisting of thousands of test cases spread over a large number of test areas. Running these large test suites in an automated manner, which execute the tests, compare and analyze the test results in sequential fashion used to take long execution time. A need was therefore felt to improve the efficiency in testing by running the tests in parallel on the target system from multiple execution platforms. Towards this objective, a new test environment called "The Integrated Test Environment for Parallel Testing" was developed which is described below.

"The Integrated Test Environment for Parallel Testing" (ITEPT) reduces test cycle-time by executing tests from multiple execution platforms concurrently. ITEPT allows the testers to execute tests from multiple execution platforms, compare, analyze the test results and generate the test reports automatically. A number of test automation tools like Test Comparator, Test Analyzer were developed to automate the entire testing process. Test Comparator compares the actual output with the expected output and Test Analyzer filters out all non-genuine mismatches and brings out only mismatches that need investigation. Each of the execution platform stores the test analysis results on a common area, which can be viewed by the user using appropriate tools.

Test Comparator is a tool, which compares the actual test output with expected test output. The test comparator generates a report identifying the places where actual test output varies from expected test output. It creates a file listing out such mismatches found between actual test output and expected test output.

Test analyzer is a tool which will take as input the output generated by test comparator and generate another report containing only genuine mismatches found between actual test output and expected test output, based on another file containing a description of mismatches that can be ignored. As an example, any date related mismatches can be ignored. Similarly a database server host name mismatching between actual test output and expected test output can be ignored. Test tool designer has to produce a file containing such mismatches, which can be ignored. There could be a number of such mismatches that can be ignored. Test analyzer will finally generate a mismatch report only for genuine mismatches found between actual test output and expected test output.

A large number of test suites have been modified to run under Integrated Test Environment for Parallel Testing. This involved analysis of a large collection of test cases and ensuring that there exist no data dependencies, functional dependencies and name conflicts among the test cases that need to be run in parallel. The measured improvement in the test execution cycles achieved for these test suites by transforming them to run under ITEPT ranged from 25% to 65%, which is a significant reduction in test execution time.

ITEPT Architecture

ITEPT is a client-server application, in which a dedicated system called the ITEPT server receives and processes every ITEPT client request. The ITEPT server provides a number of services which the ITEPT clients running on multiple client machines request. Test execution platform runs a parent process, which in turn spawns four child processes on four client machines. Client processes make request for various services to the ITEPT server, which coordinates the parallel test execution from multiple client machines by providing all necessary services.

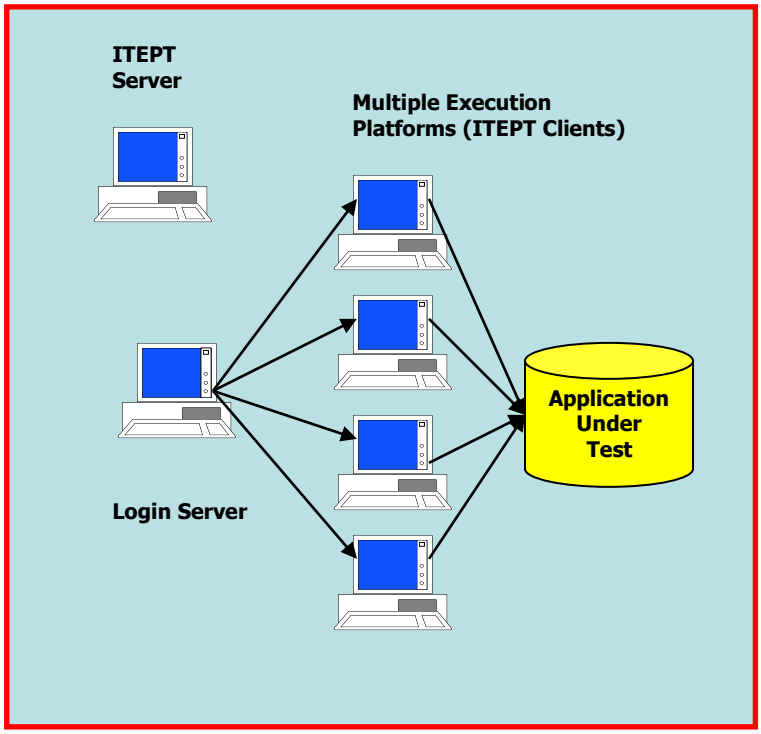

Figure 1. Integrated Test Environment for Parallel Testing Architecture

Business Benefits

- Automated test environment for parallel execution of tests
- 25% to 65% overall reduction in test execution time
- Ease of maintenance of tests and test results
- Ability to restart the tests and reusability of the tests
- Faster time to market for the product
- Reduced cost on post-release maintenance, rework
- Increased confidence in testing process and its completeness

The benefits of this test environment are obtained in each phase of software testing life cycle namely,

- Software Unit/Module/Feature Testing
- Software Feature/Subsystem Integration Testing (FIT)
- Product Integration Testing (PIT)
- System/Solution Integration Testing (SIT)
- Acceptance Testing
- Regression Testing
- Product Release Testing

Measures of Effectiveness of Test Cases

An effective way to measure the Quality of software product is the amount of code that has been tested (i.e. Code coverage). While this does not guarantee that the code is defect free, the risk of uncovering more defects from the customer's site is reduced considerably as more code is tested during the product test cycle. It should be realized that even 100% coverage does not guarantee a defect free code. Most Test engineer would agree that while one can never be sure of a bug free code, a significant milestone is achieved when "all the code has been tested." Code coverage can be a valuable measure, especially when time is taken to achieve a high coverage value.

While working with these test suites, the team took an initiative to analyze the code coverage for all the available test suites to get some degree of

confidence as to the existing level of code coverage. Code coverage provides a deep insight into the adequacy of the test cases and the need of or scope for improvement. The team undertook a comprehensive analysis of these test suites and collected code coverage data for a large number of such test suites. A Web based tool was developed where all these coverage data were stored, to order to do a comprehensive analysis of these code coverage data in various dimensions.

The Web based code coverage analysis tool developed by the team provided a convenient platform from where the user can obtain and analyze the code coverage data for various test suites. This proved to be an effective tool to quickly understand and analyze the test coverage scenarios.

The benefits of this tool were to be able to generate the following analysis reports

- Line level, function level and module level code coverage reports
- Annotated source code for function wise, module wise and test suite wise coverage data
- Annotated source code of a selected implementation file with lines hit, lines not hit, lines partially hit
- Analytical report of code coverage of a selected implementation file for various test suites
- To provide information on the most appropriate test suites to validate a bug fix/code enhancements which will guarantee the maximum statement coverage of the file being added/modified
- To analyze any field reported problems, to identify whether the root cause of the failure was due to non-coverage of the code segment where the fix for the problem was found
- To identify the root cause of any regression problems due to any limitation of existing test suites used for regression testing

Business Benefits

- An effective test tool for analysis of code coverage statistics
- Use the tool as a powerful test selection tool
- Use the tool to support white box testing for improving test coverage
- Strategic decision making in adopting appropriate test approach, test strategy, test schedule
- Achieve product stability
- Forecasting expected problems from field
- Analysis of field encountered problems
- Taking effective defect prevention measures resulting in improved product quality

Improving Effectiveness and Efficiency of Test Cases

Apart from benefits obtained through test automation and parallel test execution, the team undertook a study and analysis of the existing test suites for any possible test optimization and enhancement, which could significantly improve the effectiveness and efficiency of such test suites in terms of improved coverage and reduced execution time.

Towards this objective, the team carried out the analysis of various test suites to arrive at definitive and quantitative information about functional and code coverage of each individual test suite. The information obtained is used to identify test cases, which are redundant and do not contribute to functional and code coverage effectiveness. The idea is to identify any such overlaps of test cases across multiple test suites and eliminate them wherever possible. The second objective of this analysis was to identify the uncovered part of the application code and introduce new test cases or enhance the existing test cases to cover the uncovered code. This will led to optimization and enhancement of existing test suites, in conformity with the Optimal Cost / Enhanced Coverage model adopted for this analysis.

An exercise was carried out to evaluate code coverage performance of the existing test suites, in order to understand the nature and completeness of these test suites. Code coverage percentage for widely used test suites, for which data was collected, is given in Figure 2.

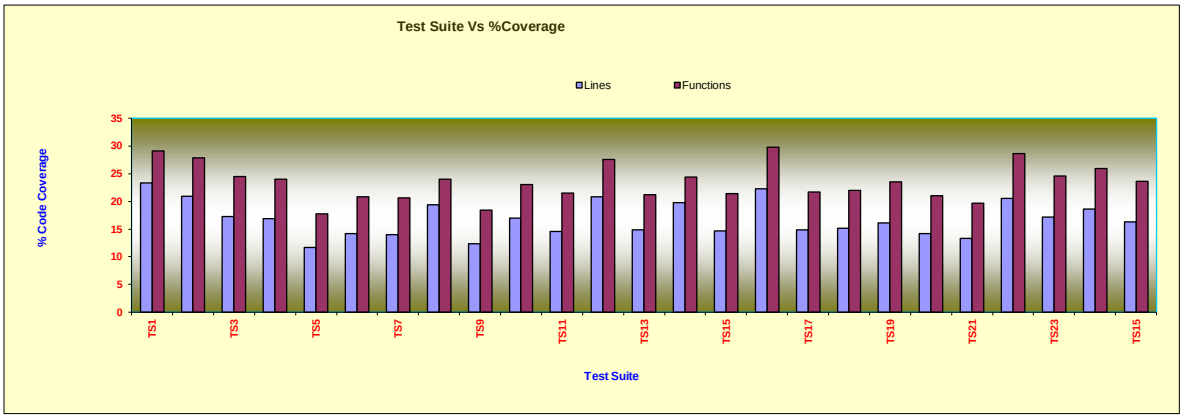

Figure 2. Code coverage of existing test suites

Statistics shown in Figure 2 signify that the code coverage of various test suites was not sufficiently high with considerable scope for improvement. Moreover, since the test suites have evolved over a period of time and have been developed by many development groups, it is likely that they contain redundant test cases resulting in increase in test execution time without necessarily contributing to the effectiveness of testing. These redundancies need to be carefully examined and removed wherever possible, without impacting the overall functionality of the test. Following section outlines the methodologies used by the team that was adopted towards this test optimization and enhancement objectives.

Optimizing and Enhancing the Test Suites

Optimal Cost – Enhanced Coverage Model:

Generally the cost parameters for optimization are

Number of Lines Covered (Ci)
Number of Test Sets (Ti)
Execution Time (Ri)

Optimal Cost - Enhanced Coverage Model optimizes
Optimal (Ti) = Enhanced (Ci) & Optimal (Ri)

Track the line coverage and reduce overlaps across the tests and arrive at minimal test sets to cover increased source code. From the code coverage analysis, the team prepared the following two cross-reference tables.

Uniqueness

The uniqueness of a test suite is defined in the present context as the unique source code covered by any existing test suite and not covered by any other test suite. It is expected that the uniqueness of all test suites should be as high as possible which is a direct measure of how optimized the test suites are. Table 1 shows some representative uniqueness across test suites.

Table 1. Uniqueness across existing test suites

Test Suite	Total Executed Functions	Unique Functions
TS #1	3836	33
TS #2	3673	59
TS #3	3229	3
TS #4	3168	4
TS #5	2332	5
TS #6	2749	5

TS #7	2718	24
TS #8	2435	0
TS #9	3039	22

Intersection

Intersection among various test suites represents the common code segment that is exercised by multiple test suites. This leads to different test suites testing the same code segment repeatedly and thereby increasing the test execution cycle time.

All such overlaps between test suites needed to be minimized by removing redundant test cases across test suites wherever applicable. Table 2 shows some representative intersection across test suites.

Table 2. Intersection among existing test suites

Test Data	TS1	TS2	TS3	TS4	TS5	TS6	TS7	TS8	TS9	TS10
TS #1	100	X	X	X	X	X	X	X	X	X
TS #2	89	100	X	X	X	X	X	X	X	X
TS #3	79	81	100	X	X	X	X	X	X	X
TS #4	75	73	80	100	X	X	X	X	X	X
TS #5	60	63	68	68	100	X	X	X	X	X
TS #6	70	74	76	77	96	100	X	X	X	X
TS #7	65	68	70	72	86	84	100	X	X	X
TS #8	78	78	82	85	98	95	86	100	X	X
TS #9	63	62	67	73	92	81	75	76	100	X
TS #10	78	79	85	79	96	90	81	85	90	100

Activities that are involved here are briefly described below.

Test Efficiency

An effective way of improving the test efficiency is optimizing the test suites through redundancy removal within a test suite and across test suites.

The following approach was followed to remove redundant test cases.

- If any test suite coverage map shows to be a complete subset of another test suite, the subset can be potentially considered for removal. If any test suite contains minimal uniqueness, corresponding unique test cases can be added to the superset and the present test suite can be dispensed with.

- Generate a coverage map of test cases for the test suites, and remove redundancy by way of removing test cases wherever coverage map matches, without impacting the overall functionality.

- Existing test suites may contain a fair amount of redundancy particularly in set-up jobs, in terms of creating identical data structures in multiple test files. It is necessary to study the setup files for every test file in a given test suite, and to combine them wherever the setup requirements match.

Generate a feature map of test cases and the feature being tested. Identify any redundancy in the feature being tested based on analysis of this map. Remove the corresponding test cases if there exists any duplication of the feature being tested. The process should be performed within a test suite as well as across test suites.

Test Effectiveness

Test effectiveness of the test cases is a measure of how comprehensive the test cases are. The test cases need to be investigated for their comprehensiveness and any lack of it needs to be

addressed by enhancing them appropriately wherever possible.

Following are the approaches that were adopted for enhancement of existing test suites. Each may contribute differently to overall improvement; nevertheless, all approaches were used.

Feature Mapping

- Create a feature map containing test scripts that map to a collection of features being tested based on examination of each test cases and relevant documentation. Use this map to identify and analyze the gaps that can be attempted to fill in.

- Prepare a cross-references table (feature map from a design perspective) of functions vs. test suites using the code coverage data available from the code coverage analysis tool for all existing test suites. Identify functions implementing specific feature and improve coverage of these functions by enhancing test cases in appropriate test area.

Coverage Mapping

- Code coverage data collected for each test case in the test suites was used for comparing against combined code coverage data for all test suites for a given source file to identify coverage gaps. Write new test cases based on the comparative coverage analysis.

In order to generate feature map and coverage map required for this analysis, test suites were executed on the test machine using ITEPT platform. A suitable code coverage tool (Rational's Purecoverage Tool) running on the test machine captures the code coverage results, which are stored into a coverage analysis database. A code coverage analysis tool generates code coverage statistical reports based on this data for various test suites. Based on these reports, necessary feature map and coverage map are generated.

The inputs from feature map and the coverage map are used in eliminating redundant test cases, adding new test cases and enhancing the existing test cases. This leads to improved test coverage and reduced test execution cycle.

Improvements attributed to newly developed test cases are computed. The process of developing new test cases, running them through ITEPT and computing the improvement is iterated till the required level of improvement is achieved.

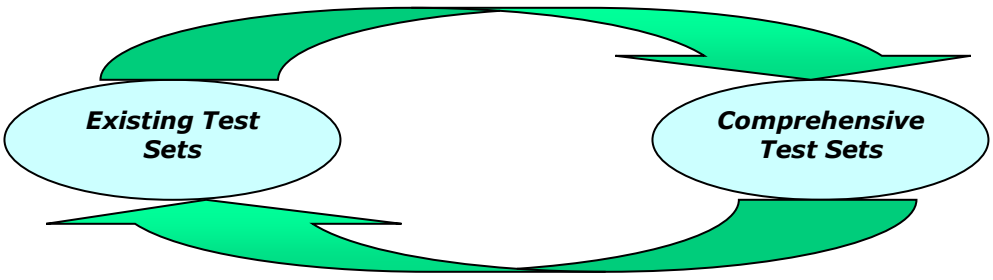

Figure 3. Test Optimization/Enhancement Process

Technical Architecture of Integrated Test Environment for Parallel Testing (ITEPT)

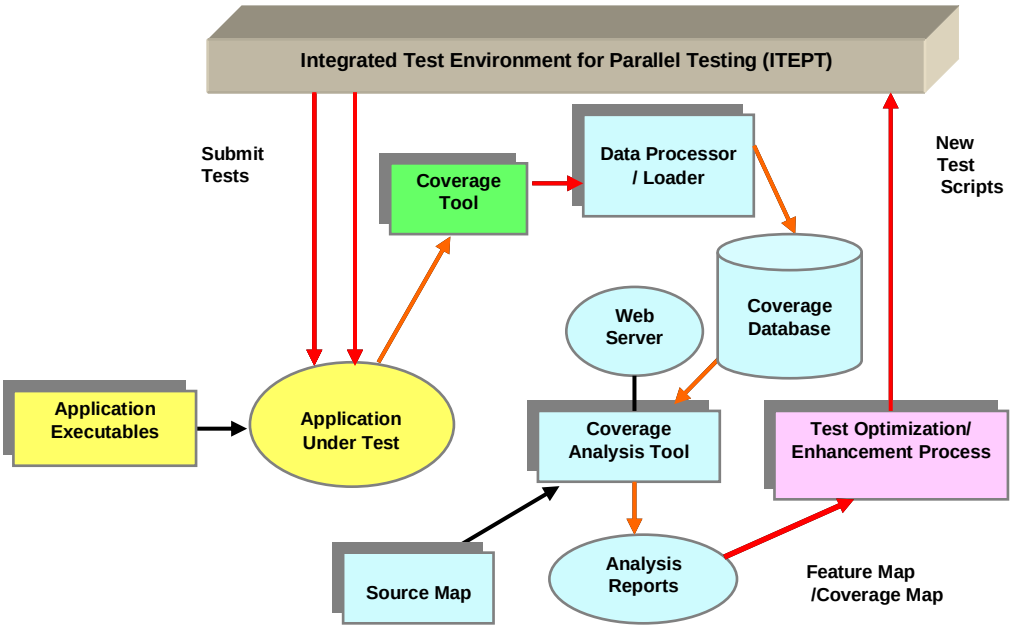

Figure 4. Technical Architecture of Integrated Test Environment for Parallel Testing (ITEPT)

Business Benefits

- More rationalized and optimized test suites
- 15% to 20% overall reduction in test execution time attributed to test rationalization
- Feature map documentation for each test suite for easy identification of test scripts and feature being tested
- Map can be used to break up test suites into smaller feature oriented groups, with resultant ease of test identification for testers
- 30% to 40% increase in code coverage of existing test suites
- Enhanced test suites with high possibility of detection of errors
- Enhanced test suites with higher quality in terms of reliability for all product releases
- Reduced cost on post-release maintenance, rework
- Increased confidence in testing process and its completeness

Optimizing Testing on Different Platforms

Following is the description of an integrated approach that can be followed to minimize testing time on different platforms. The approach can consist of

1. Baseline platform independent tests
2. Identify platform specific tests
3. Partition the test suites according to priority to arrive at minimal test set
4. Collect platform related unit test cases developed during bug fixing

Based on the above inputs, the following two models can be adopted:

Optimizing Testing for Product Release

Maintain a single set of platform independent test cases. This would be arrived at, by running this test set on all platforms successfully. This needs to be baselined at an appropriate time interval. Call this test set CT (Common Test).

Maintain a set of test cases specific to platforms. For instance, maintain sets of different test cases for various platforms like OS1, OS2…OSn. Call these platform specific test sets P1, P2…Pn respectively.

The final testing on all the above platforms will consist of Final Test Set FTS, where

FTS = CT (to be run only on any one of the platforms)
+ P1+P2+…+Pn (to be run on the respective platforms)

This approach is useful in production release testing.

Optimizing Maintenance Testing for Bug Fixes

Maintain a set of minimal test cases for each of above platforms and call them MT1, MT2…MTn.
Maintain set of recent unit test cases developed during bug fixing for each platform and call them B1, B2…Bn.

The final testing on the above platforms will consist of the Final Test Set FTS, where
FTS = (MT1+MT2+MT3+…+MTn) + (B1+B2+B3+...+Bn)

For a single platform testing, it is sufficient to run
MT(i) + B(i)

Minimal test sets will be formulated based on the high priority test cases, whose size and test execution time will be much smaller compared to original test cases.

Strengthening Unit Testing

Many of the field reported problems were traced to be due to inadequate unit testing. Moreover, it was also found that many of the regression problems have been reported due to inadequate coverage of the unit test cases used during any bug fixing and code enhancement activities. Any unit testing methodology therefore, needs to have a mechanism to evaluate and enhance the coverage of the unit test cases.

A need was therefore felt to come up with an integrated test environment that should facilitate unit testing from the developer's desktop and carry out any code coverage analysis in a convenient and automated manner. This is applicable both during application development phase and post-release maintenance phase. The unit testing platform should enable the developer to write the unit test cases, run them on the developer's build, collect code coverage data and perform any required analysis on the application files that would have been added or modified.

The unit testing tool as part of the integrated test environment should provide a platform for helping developers in unit testing by

- Automating code coverage data collection of unit test cases from the developer's desktop
- Helping the developers in setting the appropriate source map to view source code under test
- Generating analysis reports through appropriate GUI to help in ensuring that the test cases that were provided adequately cover the source code that were added/modified
- Identifying the areas of code that are not covered or partially covered in order to improve the test cases
- Using the tool as a workbench for ensuring that the test cases that were generated are adequate
- Providing an integrated unit testing environment for test engineers and developers

The unit testing will necessitate the test machine to be installed with the required application build on which unit test cases will be executed. An appropriate code coverage tool running on the test machine will capture the code coverage data for the executed unit test cases. The code coverage data thus generated can be processed and loaded into a code coverage analysis database. Web application will access the data from this code coverage analysis database and generate the necessary reports through appropriate GUI.

5. Business Benefits

- Thorough unit testing ensures robustness of the code
- A white box testing technique, to ensure high test coverage and maximum error detection

- Developer can test whether the fix that is provided is correct or not after any bug fixing
- Minimize possibility of regression problems after any bug fixing or code enhancements
- Saves time as a result of less regression problems
- Higher productivity
- Higher in-house defect detection rate
- Lower defect injection rate in coding phase
- Reduced rework due to defects
- Reduced overall customer reported problems
- High code coverage provides increased confidence to Management on test adequacy

6. Business Impact

- Automated test environment for parallel execution of tests
- More rationalized and optimized test suites
- 15% to 20% overall reduction in test execution time attributed to test rationalization
- 30% to 40% increase in code coverage of existing test suites
- Enhanced test suites with high possibility of detection of errors
- Thorough testing ensures robustness of the code
- A white box technique, to ensure high test coverage and maximum error detection
- Higher productivity
- Reduced rework due to defects
- Reduced overall customer reported problems
- Enhanced test suites with higher quality in terms of reliability for all product releases
- Minimize possibility of regression problems after any bug fixing or code enhancements

7. Cost Benefit Analysis

- Automated test environment for parallel execution of tests
- More rationalized and optimized test suites
- 15% to 20% overall reduction in test execution time attributed to test rationalization
- 30% to 40% increase in code coverage of existing test suites
- Thorough testing ensures robustness of the code
- A white box technique, to ensure high test coverage and maximum error detection
- Higher productivity
- Reduced rework due to defects
- Reduced overall customer reported problems

8. Conclusion

With test architecting as an emerging discipline and a key focus area, it should be the constant endeavor of all IT companies to lay a strong foundation for innovative test engineering practices for complex software products. In a competitive market environment like today, with increasing focus on offshore IT business model, it is imperative for IT companies to pay adequate attention in evolving effective product testing mechanisms, developing better test tools, test environment and test management. This Paper discusses a number of recommendations from a test engineering perspective for a large software product development and support activity in optimizing testing time, test effectiveness and test efficiency. It also mentions the benefits that can be derived from such innovative test engineering practices and testing framework. Even though the approach outlined here has been adopted while working with large software system maintenance, it can be extended to any software development and maintenance activities irrespective of the size, application area and domain.

9. Key References and Bibliography

1. Effective Methods for Software Testing, William E. Perry, Willey Publication
2. Software Engineering: A Practitioner's Approach, Roger S. Pressman
3. Software Testing Concepts and Tools, Nageswara Rao Pusuluri
4. Software Testing in Real Worls, Edward Kit