ISSN:0975-9646

A Suggestive Evaluation of System Test Cases in OO Systems Through Carving and Replaying Differential Unit Test Cases: A Metric Context

1. Amjan. Shaik , *CSE, Ellenki College Of Engineering and Technology(ECET), Patelguda ,Hyderabad, India.*
2. Hymavathi. Bhadriraju , *CSE, Bharath University (BU), Selaiyur, Chennai, India.*
3. K.Vikram , *CSE, City Womens College of Engineering and Technology(CWCET), Hyderabad, India.*
4. Nazeer. Shaik*, CSE, Moghal College Of Engineering and Technology(MCET),Bandlaguda, Hyderabad, India.*
5. S.V.Achuta Rao, *CSE&IT, DJR Institute of Engineering and Technology(DJRIET),Vijayawada,India.*

Abstract

In this paper an attempt is made to implement system test cases and software metrics with aid of GUI and several applications were developed to calculate the metrics and performance of the each test case, which can also be used as a stand alone method. Further an emphasis is made on different relationships with system test case and software metrics, which will helps to determine quality and quantity of software attributes measured with regard of Object-Oriented Software Development Life Cycle. We demonstrate a suggestive evaluation of system test cases in OO Systems. Developing effective suites of unit test cases presents a number of challenges. Specifications of unit behavior are usually informal and are often incomplete or ambiguous, leading to the development of overly general or incorrect unit tests. Furthermore, such specifications may evolve independently of implementations requiring additional maintenance of unit tests even if implementations remain unchanged. Testers may find it difficult to imagine sets of unit input values that exercise the full range of unit behavior and thereby fail to exercise the different ways in which the unit will be used as a part of a system. Unit test cases are focused and efficient. System tests are effective at exercising complex usage patterns. Differential unit tests (DUTs) are a hybrid of unit and system tests that exploits their strengths. They are generated by carving the system components, while executing a system test case that influence the behavior of the target unit and then reassembling those components so that the unit can be exercised as it was by the system test. Here, we show that DUTs retain some of the advantages of unit tests, can be automatically generated, and have the potential for revealing faults related to intricate system executions. We describes a framework for carving and replaying DUTs that accounts for a wide variety of strategies and trade-offs, we implement an automated instance of the framework with several techniques to mitigate test cost enhance flexibility and robustness. The goal of this paper is to empirically explore the relationship between OOD Metrics with Test Cases .We empirically analyzed and tested with Open Source Java projects[11,12,13].

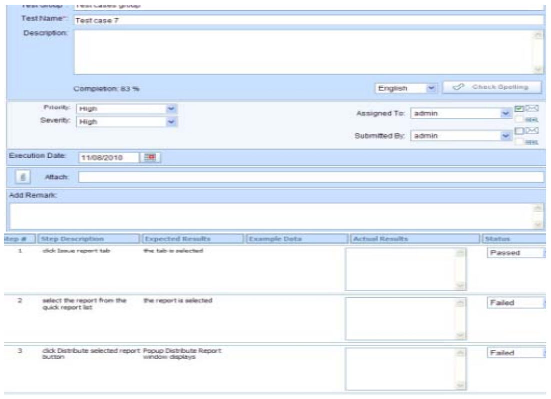
Keywords: Unit Testing, Test Case, Software Metrics, Regression Testing.

I. INTRODUCTION

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing also provides an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include, the process of executing a program or application with the intent of finding software bugs (errors or other defects).An empirical study performed on different open source java based projects. PCA (Principal Components Analysis) method was used to perform this evaluation Object-Oriented design and development is becoming very popular in today's software development environment. Object Oriented development requires not only a different approach to design and implementation, it requires a different approach to software metrics. Since Object-Oriented technology uses objects and not algorithms as its fundamental building blocks, the approach to software metrics for Object-Oriented programs must be different from the standard metrics set. However, it is not apparent for a developer or a project manager to select the metrics that are more useful. Furthermore, these metrics are not completely independent. Using several metrics at the same time is time consuming and can generate a quite large data set, which may be difficult to analyze and interpret. These days Object-Oriented Metrics emerged as adequate in several domains of Software Engineering [1, 5]. Various parameters in connection with the software products and processes are assessed through the use of Software Metrics. By applying these metrics to software, it becomes possible to gather numerical data that quantifiable, related to context dimensions. The ability of external elements like rely, testing and maintenance of software influence the accuracy of the resultant values of metrics [2, 7]. Metrics are then used to predict software quality [3].Software engineers develop unit test cases to validate individual program units such as methods, classes, and packages, before they are integrated into the whole system. By focusing on an isolated unit, unit tests are not constrained or influenced by other parts of the system in exercising the target unit. This smaller scope for testing usually results in more efficient test execution and fault isolation relative to full system testing and debugging. Unit test cases are also key components of several development and validation methodologies, such as extreme programming (XP), test-driven development (TDD) practices, continuous testing, efficient test prioritization and selection techniques. Specifications of unit behavior are usually informal and are often incomplete or ambiguous, leading to the development of overly general or incorrect unit tests. Furthermore, such specifications may evolve independently of implementations requiring additional maintenance of unit tests even if implementations remain unchanged. Testers may find it difficult to imagine sets of unit input values that exercise the full range of unit behavior and thereby fail to exercise the different ways in which the unit will be used as a part of a system. An alternative approach to unit test development, which does not rely on specifications, is based on the analysis of a unit's implementation. Testers developing unit tests on achieving coverage-adequacy criteria in testing the target unit's code. Such tests are inherently susceptible to errors of exception with respect to specified unit behavior and may thereby miss certain faults. Finally, unit testing requires the development of test harnesses or the setup of a testing framework (e.g. JUnit) to make the units executable in isolation. Software engineers also develop system tests, usually based on documents that are available for most software systems that describe the system's functionality from the user's perspective ie, requirement documents and user's manuals. This makes system tests appropriate for determining the readiness of a system for release or its acceptability to customers[7]. Additional benefits accrue from testing system-level behaviors directly. First, system tests can be developed without an intimate knowledge of the system internals, which reduces the level of expertise required by test developers and makes tests less sensitive to implementation-level changes that are behavior preserving. Second, system tests may expose faults that unit tests do not, for example, faults that emerge only when multiple Units are integrated and jointly utilized. Finally they involve executing the entire system, no individual harnesses need to be constructed. System tests are an essential component of all practical software validation methods. Fault isolation and repair during system testing can be significantly more expensive than during unit testing.

II. RESEARCH BACKGROUND

Written specifications and user documentation can provide you with excellent information for making test cases. Later, you can write more test cases based on the function and flow of the application. At this point, you are ready to group test cases together to form a test procedure. Finally, you can automate the running of test cases for regression testing. This way the testers and others in QA can work on checking new functionality. Fields that commonly happen in test cases are: Test case ID, Unit to test, Assumptions, Test data, Steps to be executed, Expected result, Actual result, Pass/Fail, Comments



Test Group: Test cases group
Test Name*: Test case 7
Description:
Completion: 83 %
English
Check Spelling
Priority: High
Severity: High
Assigned To: admin
Submitted By: admin
Execution Date: 11/08/2010
Attach:
Add Remark:

Step #	Step Description	Expected Results	Example Data	Actual Results	Status
1	click Issue report tab	the tab is selected			Passed
2	select the report from the quick report list	the report is selected			Failed
3	click Distribute selected report button	Popup Distribute Report window displays			Failed

Figure 1: Test cases

In the literature, software engineers develop unit test cases to validate individual program units like methods, classes, and packages, before they are integrated into the whole system. By focusing on an isolated unit, unit tests are not constrained or influenced by other parts of the system in exercising the target unit. Software engineers also develop system tests, usually based on documents that are available for most software systems that describe the system's functionality from the user's perspective, ie, requirement documents and user's manuals. This makes system tests appropriate for determining the readiness of a system for release or its acceptability to customers.

Demerits:

1. They can be expensive to execute, for large systems, days or weeks, and considerable human effort may be needed for running a thorough suite of system tests.
2. System testing may fail to exercise the full range of behavior implemented by a system's particular units, thus system testing cannot be viewed as an effective replacement for unit testing.

In the proposed Methodology, DUTs are created from system tests by capturing components of the exercised system that may influence the behavior of the targeted unit and that reflect the results of executing the unit; we term this carving because it involves extracting the relevant parts of the program state corresponding to the components exercised by a system test. Those components are automatically assembled into a test harness that establishes the pre-state of the unit that was encountered during system test execution. From that state, the unit is replayed and the resulting state is queried to determine if there are differences with the recorded unit post state.

Merits:

1.We improve the cost and effectiveness of system tests and carved unit tests.

2.The results indicate that carved test cases can be as effective as system test cases in terms of fault
detection, but much more efficient in the presence of localized changes.

3.A framework for automatically carving and replaying DUTs that accounts for a wide variety of implementation strategies with different trade-offs.

4.Object Oriented Design Metrics measures effectively at Design and Testing level.

III. DESIGN METHODOLOGY

The most creative and challenging phase of the life cycle is system design. The term design describes a final system and the process by which it is developed. It refers to the technical specifications that will be applied in implementations the candidate system. The design may be defined as "the process of applying various techniques and principles for the purpose of defining a device, a process or a system in sufficient details to permit its physical realization".The goal of designer is First, how the output is to be produced and in what format samples of the output and input are also presented. Second, input data and database files have to be designed to meet the requirements of the proposed output. The processing phases are handled through the program Construction and Testing. Finally, details related to justification of the system and an estimate of the impact of the candidate system on the user and the organization are documented and evaluated by management as a step toward implementation.

The importance of software design can be stated in a single word *"Quality"*. Design provides us with representations of software that can be assessed for quality. Design is the only way that we can accurately translate a customer's requirements into a finished software product or system without design we risk building an unstable system, that might fail it, small changes are made or may be difficult to test, or one who's quality can't be tested. So it is an essential phase in the development of a software product.

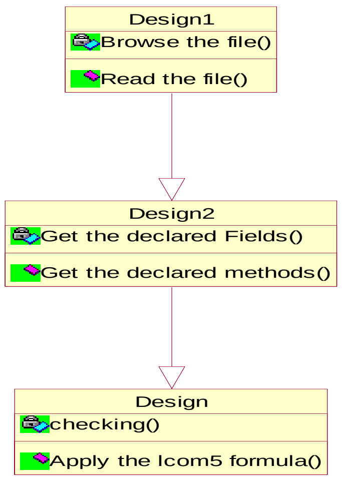

Design1
Browse the file()
Read the file()
Design2
Get the declared Fields()
Get the declared methods()
Design
checking()
Apply the Icom5 formula()

Figure 1 Class Diagram

IV. SYSTEM ARCHITECTURE

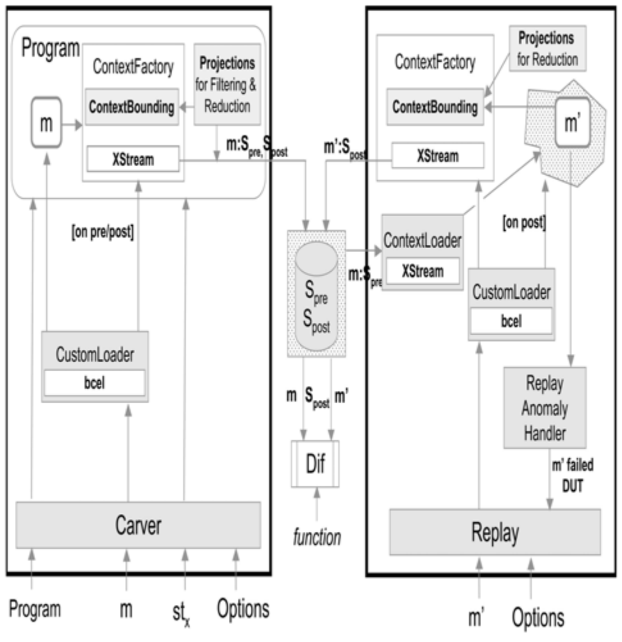


Figure 2 :CR Tool Architecture

4.1 Process Model

Process models define a distinct set of activities , actions, tasks, milestones, and work products that are required to engineer for high-quality software. They provide a useful road-map for software engineering work[3,4].The Classic Life Cycle, Suggests a systematic, sequential approach to software development that begins with customer specification of requirements and progresses through Planning, Modeling, Construction, and Deployment, culminating in on-going support of the complete software.

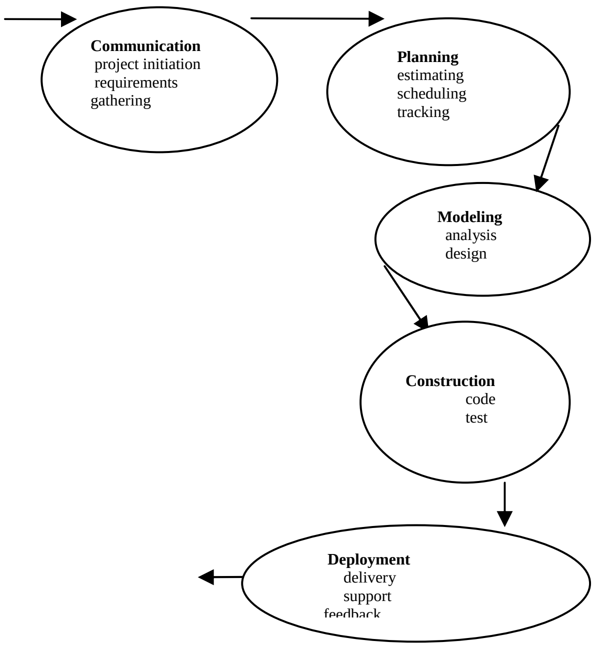


Figure 3: Classic Life Cycle

V. FRAMEWORK FOR TEST CARVING AND REPLAY

The development stage takes as its primary input the design elements described in the approved design document. For each design element, a set of one or more software artifacts will be produced, appropriate test cases will be developed for each set of functionally related software artifacts, and an online help system will be developed to guide users in their interactions with the software.

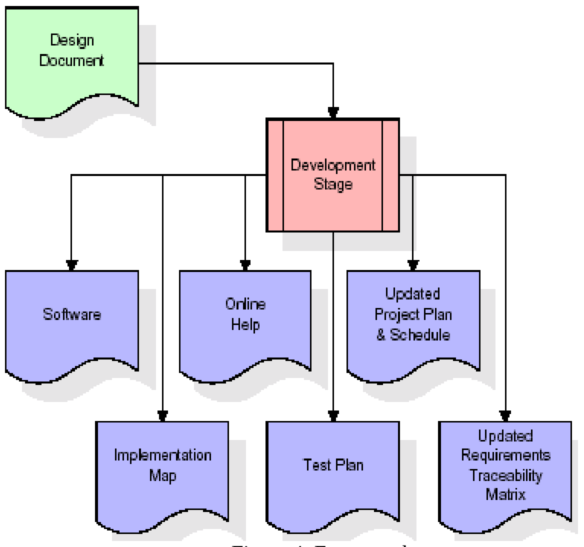


Figure 4: Framework

At this point, the RTM is in its final configuration. The outputs of the development stage include a fully functional set of software that satisfies the requirements

and design elements previously documented, an online help system describes the test cases to be used to validate the correctness and completeness of the software, an updated RTM, and an updated project plan will have:
1. Identify a program state from which to initiate testing,
2. Establish that program state,
3. Execute the unit from that state, and
4. Judge the correctness of the resulting state.

Improving CR with Projections:
We focus CR testing on a single method by defining projections on carved pre-sates that preserve information related to the unit under test and are likely to provide significant reduction in pre-state size.
Instantiation of the framework:
The carving activity starts with the Carver class which takes four inputs: the program name, the target method(s) m within the program, the system test case stx inputs, the reduction and filtering options.
Clustering projection.:
The clustering projection attempts to identify a set of similar UTs,DUTxcallee;1;DUTx!callee;2; . . . , that result from the repeated invocation of callee from within the same DUT, DUTx caller, of method caller.
Evaluating the framework:
Efficiency: We first focus on the efficiency of the carving process. Although our infrastructure completely automates carving, this process does consume time and storage so it is important to assess its efficiency as it might impact its adoption and scalability.
Fault detection effectiveness: Most of the test suites carved from S-selection, (with k _ 1), C-selection mayref ,and C-selection-touched detected as many faults as the S-retest-all technique. This indicates that a DUT test suite can be as effective as a system test suite at detecting faults, even when using aggressive projections.

Necessity for Software Metrics

Now days lot of software's are developed by the developers. Many of the software's are very big in code size. So generally to maintain the quality of the code, developers need to is tribute the code in small pieces or parts. But how to divide the software is also an important task as it can lead to various problem of inter module communication therefore this modularized code should also be checked for the quality. There are problems in removing the errors of non modularized code. Particularly in object oriented software development developer needs to use a lots of object oriented concepts which may introduced the inter dependency of the various units of the software e.g. Inheritance. A software metric is a measure of some property of a piece of software or its specifications. Therefore software metrics suite is needed [12]. We are concentrating on the same issue and providing the software metrics for this modularized object oriented code.

5.1Framework Approach in Automation

A framework is an integrated system that sets the rules of Automation of a specific product. This system integrates the function libraries, test data sources, object details and various reusable modules. These components act as small building blocks which need to be assembled in a regular fashion to represent a business process. Thus, framework provides the basis of test automation and hence simplifying the automation effort. There are various types of frameworks. They can be categorized on the basis of the automation component they leverage. They are:
1. Data-driven testing
2. Modularity-driven testing
3. Keyword-driven testing
4. Hybrid testing
5. Model-based testing

Regression Testing
Regression testing is a type of software testing that seeks to uncover software regressions. Such regressions occur whenever previously working software functionality stops working as intended. Typically, regressions occur as an unintended consequence of program changes. *Common methods* of regression testing include rerunning previously run tests and checking whether previously fixed faults have re-emerged[10,11].
Experience has shown that as software is fixed, emergence of new and reemergence of old faults is quite common. Sometimes reemergence occurs because a fix gets lost through poor revision control practices . Often, a fix for a problem will be "fragile" in that it fixes the problem in the narrow case where it was first observed but not in more general cases which may arise over the lifetime of the software. Frequently, a fix for a problem in one area inadvertently causes a software bug in another area. Finally, it has often been the case that when some feature is redesigned, the same mistakes that were made in the original implementation of the feature were made in the redesign. Therefore, in most software development situations it is considered good practice that when a bug is located and fixed, a test that exposes the bug is recorded and regularly retested after subsequent changes to the program. Although this may be done through manual testing procedures using programming techniques, it is often done using automated testing tools. Such a test suite contains software tools that allow the testing environment to execute all the regression test cases automatically; some projects even set up automated systems to automatically re-run all regression tests at specified intervals and report any failures. Common strategies are to run such a system after every successful compile, every night, or once a week. Those strategies can be automated by an external tool, such as BuildBot.
Regression testing is an integral part of the extreme programming software development method. In this method, design documents are replaced by extensive, repeatable, and automated testing of the entire software package at every stage in the software development cycle. Traditionally, in the corporate world, regression testing has been performed by a software quality assurance team after the development team has completed work. However, defects found at this stage are the most costly to fix. This problem is being addressed by the rise of developer testing. Although developers have always written test cases as part of the development cycle, these test cases have generally been either

functional tests or unit tests that verify only intended outcomes. Developer testing compels a developer to focus on unit testing and to include both positive and negative test cases[1].

Merits

Regression testing can be used not only for testing the *correctness* of a program, but often also for tracking the quality of its output. For instance, in the design of an application, regression testing should track the code size, simulation time and time of the test suite cases. Also as a consequence of the introduction of new bugs, program maintenance requires far more system testing per statement written than any other programming. Theoretically, after each fix one must run the entire batch of test cases previously run against the system, to ensure that it has not been damaged in an obscure way.

Regression analysis

Regression Analysis includes any techniques for modeling and analyzing several variables, when the focus is on the relationship between a Dependent variable, one or more Independent variables. More specifically, regression analysis helps us understand how the typical value of the dependent variable changes when any one of the independent variables is varied, while the other independent variables are held fixed. Most commonly, regression analysis estimates the conditional expectation of the dependent variable given the independent variables ie, the average value of the dependent variable when the independent variables are held fixed. Less commonly, the focus is on a quantile, or other location parameter of the conditional distribution of the dependent variable given the independent variables. In all the aces, the estimation target is a function of the independent variables called the *regression function*. In regression analysis, it is also of interest to characterize the variation of the dependent variable around the regression function, which can be described by a probability distribution.

Regression analysis is widely used for prediction including forecasting of time-series data. Use of regression analysis for prediction has substantial overlap with the field of machine learning. Regression analysis is also used to understand which among the independent variables are related to the dependent variable, and to explore the forms of these relationships. In restricted circumstances, regression analysis can be used to infer causal relationships between the independent and dependent variables. A large body of techniques for carrying out regression analysis has been developed. Familiar methods such as linear regression and ordinary least squares regression are parametric, in that the regression function is defined in terms of a finite number of unknown parameters that are estimated from the data. Nonparametric regression refers to techniques that allow the regression function to lie in a specified set of functions, which may be infinite-dimensional. The performance of regression analysis methods depends on the form of the data-generating process, and how it relates to the regression approach being used. Since the true form of the data-generating process is not known, regression analysis depends to some extent on making assumptions about this process. These assumptions are sometimes but not always testable, if a large amount of data is available. Regression models for prediction are often useful even when the assumptions are moderately violated, although they may not perform optimally[8]. However when carrying out inference using regression models, especially involving small effects or questions of causality based on observational data, regression methods must be used cautiously as they can easily give misleading results.

Underlying Assumptions

Classical assumptions for regression analysis include:

- The sample must be representative of the population for the inference prediction.
- The error is assumed to be a random variable with a mean of zero conditional on the explanatory variables.
- The independent variables are error-free. If this is not so ,modeling may be done using errors-in-variables model techniques.
- The predictors must be linearly independent, i.e. it must not be possible to express any predictor as a linear combination of the others.
- The errors are uncorrelated, that is, the variance-covariance matrix of the errors is diagonal and each non-zero element is the variance of the error.
- The variance of the error is constant across observations . If not, weighted least squares or other methods might be used.

These are sufficient but not all necessary conditions for the least-squares estimator to possess desirable properties, in particular, these assumptions imply that the parameter estimates will be unbiased, consistent, and efficient in the class of linear unbiased estimators. Many of these assumptions may be relaxed in more advanced treatments.

VI. SOFTWARE TESTABILITY

IEEE defines testability as the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met. ISO defines testability as attributes of software that bear on the effort needed to validate the software product. Testability is defined as an important characteristic of maintainability. In order to help in appraising the ease (or difficulty) for testing software, many testability analysis and measurement approaches have been proposed these last several years. These approaches were investigated within different application domains.

Fenton et *al.* define testability as an external attribute. Freedman introduced testability measures for software components based on two factors: observability and controllability. He defined observability as the ease of determining if specific inputs affect the outputs of a component, and controllability as the ease of producing specific outputs from specific inputs. The introduced testability measures are only applied to functional specifications by examining input and output domains. Voas defines testability as the probability that a test case will fail if the program has a fault [6]. He considers that testability is the combination of the probability that a location is executed, the probability of a fault at a location, and the probability that corrupted results will propagate to the observable outputs. Voas and Miller propose a testability metric based on the inputs and outputs domains of a software component, and the PIE (Propagation, Infection and Execution) technique to analyze software testability [2].

Binder [13] discusses software testability based on six factors: representation, implementation, built-in text, test suite, test support environment and software process capability. Khoshgoftaar et al. [3] modeled the relationship between static software product measures and testability. They used the developed model to classify the component program modules as having low or high testability. They used the fault-based definition of testability proposed by Voas et al. [6]. Software testability is considered as a probability predicting whether tests will detect a fault. Khoshgoftaar et al. [4] applied neural networks to predict testability from static software metrics.

McGregor et al. [5] address testability of OOS and introduce the visibility component measure (VC). Bertolino et *al.* [11] investigate the concept of testability and its use in dependability assessment. They adopt a definition of testability, as a conditional probability, somewhat different from the one proposed by Voas et *al.* [6]. They derive the probability of program correctness using a Bayesian inference procedure. Le Traon et *al.* [4, 7, 8] propose testability measures for dataflow designs. Petrenko et *al.* [5] and Karoui et *al.* address testability in the context of communication software. Sheppard et *al.* [5] focuses on formal foundation of testability metrics. Jungmayr [3] investigates testability measurement based on static dependencies within OOS. He takes an integration testing point of view and uses this approach to identify test-critical dependencies.

Gao et *al.* [3] consider testability from the perspectives of component-based software construction. They define component testability based on five factors: understandability, observability, controllability, traceability and testing support capability. They argue that component testability can be verified and measured based on the five factors in a quality control process. According to Gao et *al.*, software testability is not only a measure of the effectiveness of a test process, but also a measurable indicator of the quality of a software development process. They address component testability issues by introducing a model for component testability analysis during a component development process.

Nguyen et al. [5] focused on testability analysis based on data flow designs in the context of embedded software. Baudry et al. [9] addressed testability measurement of object-oriented designs. They focused on design patterns as coherent subsets in the architecture, and explained how their use can provide a way for limiting the severity of testability weaknesses. A testability measurement for UML class diagrams is proposed. They detect undesirable configurations in UML class diagrams, which they call testability anti-patterns. They also proposed solutions to improve the testability of the design [10].

Metrics can be used to assess software testability. Metrics can, in fact, be used to locate parts of a program which contribute to a lack of testability. Bruntink et al. [11] investigate factors of the testability of OOS. They evaluated a set of well-known object-oriented metrics with respect to their capabilities to predict testability of classes of a Java system. They investigate testability from the perspective of unit testing. More recently, Chowdhary [12] focuses on why it is so difficult to practice testability in the real world.

Software Testing Metrics

1. Cost of finding a defect in testing (CFDT) = Total effort spent on testing / defects found in testing [Total time spent on testing including time to create, review, rework, execute the test cases and record the defects. This should not include time spent in fixing the defects].

2. Test Case Adequacy: This defines the number of actual test cases created vs estimated test cases at the end of test case preparation phase. It is calculated as No. of actual test cases / No: of test cases estimated

3. Test Case Effectiveness: This defines the effectiveness of test cases which is measured in number of defects found in testing without using the test cases. It is calculated asNo. of defects detected using test cases*100/Total no: of defects detected

4. Effort Variance can be calculated as {(Actual Efforts-Estimated Efforts) / Estimated Efforts} *100

5. Schedule Variance: It can be calculated as {(Actual Duration - Estimated Duration)/Estimated Duration}*100

6. Schedule Slippage: Slippage is defined as the amount of time a task has been delayed from its original baseline schedule. The slippage is the difference between the scheduled start or finish date for a task and the baseline start or finish date. It is calculated as ((Actual End date - Estimated End date) / (Planned End Date – Planned Start Date) * 100

7. Rework Effort Ratio:

{(Actual rework efforts spent in that phase / Total actual efforts spent in that phase)} * 100

8. Review Effort Ratio:

(Actual review effort spent in that phase / Total actual efforts spent in that phase) * 100

9. Requirements Stability Index:

{1 - (Total No. of changes /No of initial requirements)}

10. Requirements Creep:

(Total No. of requirements added / No of initial requirements) * 100

11. Weighted Defect Density:

WDD = (5*Count of fatal defects)+(3*Count of Major defects)+(1*Count of minor defects)

Here the Values 5, 3, 1 correspond to severities as mentioned below:

Fatal-5

Major-3

Minor-1

12.**The Defect Removable Efficiency (DRE)** is the

percentage of defects that have been removed during an activity, computed with the equation :
DRE = (Number of Defects Removed / Number of Defects at Start of Process) * 100
The DRE can also be computed for each software development activity and plotted on a bar graph to show the relative defect removal efficiencies for each activity. Or, the DRE may be computed for a specific task or technique (e.g. design inspection, code walkthrough, unit test, 6 month operation, etc.),We can also calculate DRE as: DRE = A / (A+B)
where A = Defects by raised by testing team and B = Defects raised by customer
If dre <=0.8 then good product otherwise not.

Test Phase Metrics

For all projects the following metrics will be captured and published by the QA team during the Test Phase.
Metrics that look at Functional Areas/Requirements check for test coverage and consistency of test effort.

Test Process Metrics

The following are provided during the Test Preparation stage of the Test Phase:

• Test Preparation

- Number of Test Requirements Vs Functional Areas/Requirements (Test coverage)
- Number of Test Cases Planned Vs Ready for Execution
- *Total Time Spent on Preparation Vs Estimated Time*

The following are provided during the Test Execution stage of the Test Phase:

• Test Execution and Progress

- Number of Test Cases Executed Vs Test Cases Planned
- Number of Test Cases Passed, Failed and Blocked
- Total Number of Test Cases Passed by Functional Areas/Requirements
- Total Time Spent on Execution Vs Estimated Time

Test Product Metrics

• Bug Analysis

- Total Number of Bugs Raised and Closed per Period
- Total Number of Bugs Closed Vs Total Number of Bugs Re-Opened (Bounce Rate)
- Bug Distribution Totals by Severity per Period
- Bug Distribution Totals by Functional Areas/Requirements by Severity per Period.

VII. EMPIRICAL RESULTS

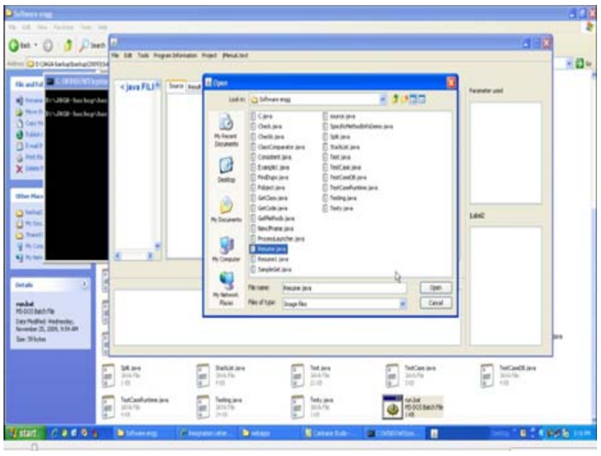
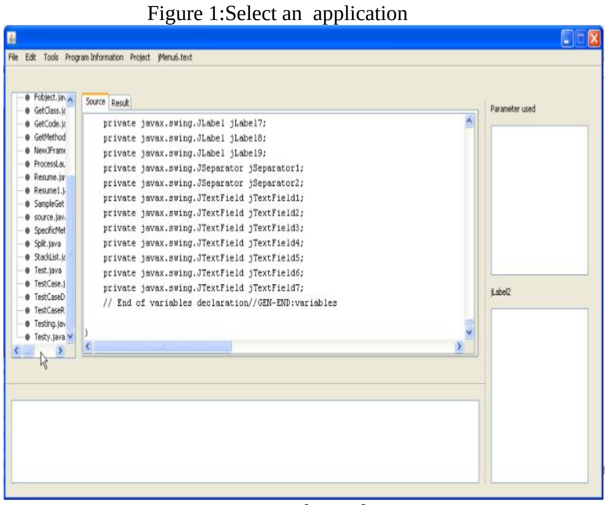


Figure 1:Select an application



```
    private javax.swing.JLabel jLabel7;
    private javax.swing.JLabel jLabel8;
    private javax.swing.JLabel jLabel9;
    private javax.swing.JSeparator jSeparator1;
    private javax.swing.JSeparator jSeparator2;
    private javax.swing.JTextField jTextField1;
    private javax.swing.JTextField jTextField2;
    private javax.swing.JTextField jTextField3;
    private javax.swing.JTextField jTextField4;
    private javax.swing.JTextField jTextField5;
    private javax.swing.JTextField jTextField6;
    private javax.swing.JTextField jTextField7;
    // End of variables declaration//GEN-END:variables

}
```

Figure 2:Execute the application

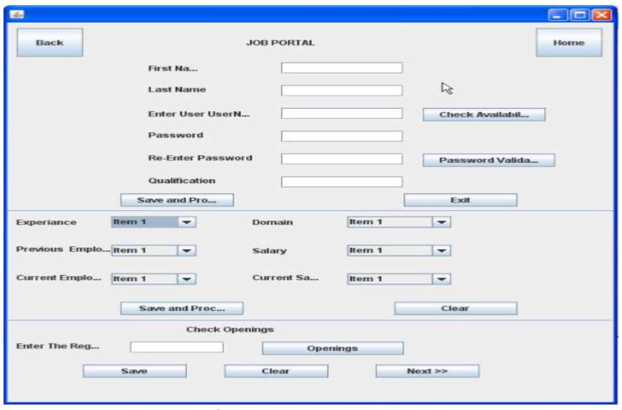
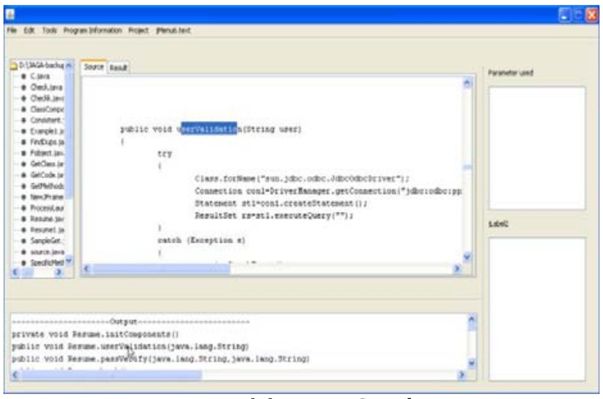


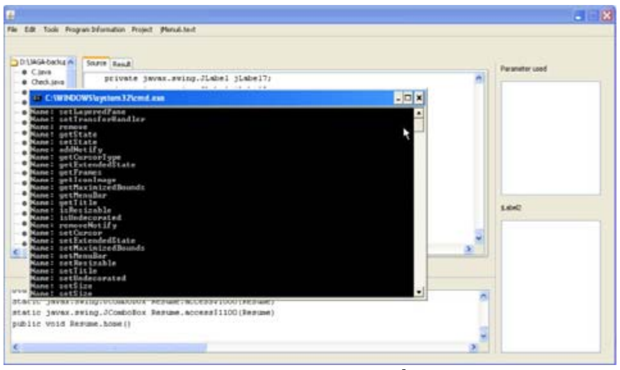
Figure 3:Considering test case on various parameters



```
        public void userValidation(String user)
        {
                try
                {
                        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                        Connection con1=DriverManager.getConnection("jdbc:odbc:pp
                        Statement st1=con1.createStatement();
                        ResultSet rs=st1.executeQuery("");
                }
                catch (Exception e)
                {
```

```
--------------------Output------------------------
private void Resume.initComponents()
public void Resume.userValidation(java.lang.String)
public void Resume.passVerify(java.lang.String,java.lang.String)
```

Figure 4:Validations on Specific parameters

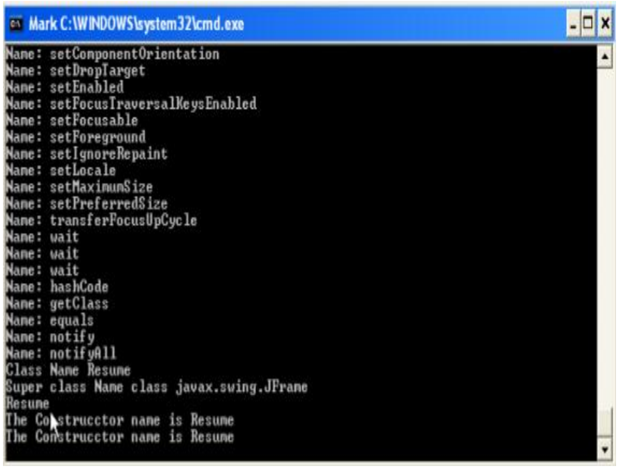


```
    private javax.swing.JLabel jLabel7;
```

```
Name: setLayeredPane
Name: setTransferHandler
Name: remove
Name: getState
Name: setState
Name: addNotify
Name: getCursorType
Name: getExtendedState
Name: getFrames
Name: getIconImage
Name: getMaximizedBounds
Name: getMenuBar
Name: getTitle
Name: isResizable
Name: isUndecorated
Name: removeNotify
Name: setCursor
Name: setExtendedState
Name: setMaximizedBounds
Name: setMenuBar
Name: setResizable
Name: setTitle
Name: setUndecorated
Name: setSize
Name: setSize
```

```
static javax.swing.JComboBox Resume.access$1100(Resume)
public void Resume.home()
```

Figure 5: Test case results



```
Name: setComponentOrientation
Name: setDropTarget
Name: setEnabled
Name: setFocusTraversalKeysEnabled
Name: setFocusable
Name: setForeground
Name: setIgnoreRepaint
Name: setLocale
Name: setMaximumSize
Name: setPreferredSize
Name: transferFocusUpCycle
Name: wait
Name: wait
Name: wait
Name: hashCode
Name: getClass
Name: equals
Name: notify
Name: notifyAll
Class Name Resume
Super class Name class javax.swing.JFrame
Resume
The Construcctor name is Resume
The Construcctor name is Resume
```

Figure 6:Validations on a package

ACKNOWLEDGEMENTS

The authors thankful to the M.Tech (CSE) students, Bharath University,Chennai for the preparation of this document as part of their M.Tech Project and also thankful to Ellenki College of Engineering & Technology‘s Research & Development Center, Hyderabad for their cooperation and support in data collection.

CONCLUSION

The main advantage of this new model being proposed is that it unifies the various OOS attributes and helps to capture much more than the simple static structure of a system. In order to ensure application of this model in a more generalized manner, we need to replicate this study on other large projects in addition to assessing the validity of the model for predicting the testability, fault proneness and maintainability. We presented tools and techniques that allow us to dynamically collect stacks in multithreaded GUI applications, .including entries from the libraries that they use. In addition , we empirically demonstrated the feasibility and effectiveness of using dynamically collected call stacks as a coverage criterion for GUI applications. We have shown that event-driven GUI applications are sufficiently different from traditional applications to require new coverage criteria. In our future work, we plan to further generalize our results for coverage criteria that are effective for GUI testing scenarios. Although we were able to successfully analyze complete call stack coverage data for the TerpOffice applications, that data volume for even larger applications may become unwieldy. Thus, we intend to look for techniques that reduce the number of coverage requirements generated by a complete call stack data collection while still retaining call stack coverage's desirable qualities .One idea is to limit the depth of calls into library routines. Another strategy is to define a similarity metric for call stacks such that different stacks with a certain similarity value may be considered redundant and therefore be discarded. A large number of object-oriented (OO) metrics are used to assess different software attributes. Software metrics can be calculated automatically from source code. The assessment of even large software systems can be performed quickly at a low cost. Software metrics can be useful in predicting software quality attributes and supporting various software engineering activities. Empirical validation of software metrics is therefore important to ensure their practical relevance. Metrics can be used to assess software testability. Metrics can be used to locate parts of a program which contribute to a lack of testability. Bruntink et al. investigate factors of the testability of OOS. They evaluated a set of well-known object-oriented metrics with respect to their capabilities to predict testability of classes of a Java system. They investigate testability from the perspective of unit testing.

REFERENCES

[1] J. Bach. Useful features of a test automation system *Testing Techniques Newsletter*, Oct. 1996.

[2] K. Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston,
MA, USA, 2002.

[3] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*, chapter 18, pages 943–951. Object Technologies. Addison Wesley, Reading, Massachusetts, USA, first edition, 1999.

[4] D. Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8):498–516,Aug. 1997

[5] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *International Symposium on Software Testing and Analysis*, pages 123–133, 2002.

[6] L. C. Briand, M. D. Penta, and Y. Labiche. Assessing and improving state-based class testing: A series of experiments. *IEEE Trans. Software. Engineering*, 30(11):770–793, 2004.

[7] Y. Chen, D. Rosenblum, and K. Vo. Test Tube: A system for selective regression testing. In *Proc. of the 16th Int Conf.on Software Engineering*, pages 211–220, May 1994.

[8] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The jml and junit. In *European Conference on Object-Oriented Programming*, pages 231–255, June 2002.

[9] C. Csallner and Y. Smaragdakis. Jcrasher: an automatic robustness tester for java. *Softw. Pract. Exper.*, 34(11):1025–1050, 2004.

[10] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435,2005.

[11] Amjan Shaik,Dr.C.R.K Reddy"Statiscal Analysis for Object Oritented Design Software Metrics", International Journal of Engineering Science and Technology(IJEST),Vol. 2(5), 2010, 1136-1142.

[12] Amjan Shaik, C. R. K. Reddy, Bala Manda, Prakashini. C, Deepthi. K" An Empirical Validation of Object Oriented Design Metrics in Object Oriented Systems" International Journal of Emerging Trends in Engineering and Applied Sciences (IJETEAS) 1 (2): 216-224 (ISSN: 2141-7016).

[13] Amjan Shaik, C. R. K. Reddy, Bala Manda, Prakashini. C, Deepthi," Metrics for Object Oriented Design Software Systems: A Survey "International Journal of Emerging Trends in Engineering and Applied Sciences (IJETEAS) 1 (2): 190-198 (ISSN: 2141-7016).

[14] Aggarwal, K.K., Yogesh, S., Arvinder, K., and Ruchika, M., "Empirical study of object-oriented metrics", Journal of Object Technology, vol. 5, no. 8, 2006.

[15] Aggarwal, K.K., Yogesh, S., Arvinder, K., and Ruchika, M., "Empirical analysis for investigating the effect of object-oriented metrics on fault proneness: A replicated case study", *Software Process: Improvement and Practice*, 16 (1), 2009.

[16] Bruntink, M., and Deursen, A.V., "Predicting Class Testability using Object-Oriented Metrics", Fourth Int. Workshop on Source Code Analysis and Manipulation (SCAM), IEEE Computer Society, 2004.

[17] Bruntink, M., and Van Deursen, A., "An empirical study into class testability". *Journal of Systems and Software*, 79, 9, 2006.

[18] McGregor, J., and Srinivas, S., "A measure of testing effort", Proceeding of the Conference on Object-Oriented Technologies, pages 129-142. USENIX Association, June1996.

[19] Zhou, Y., and Leung, H., "Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults". IEEE TSE, Vol. 32, No. 10, October 2006.

[20] Sommervile, I., "Software Engineering", Addison-Wesley, 2007.

ABOUT THE AUTHORS

Amjan Shaik is working as a Professor and Head, Department of Computer Science and Engineering at Ellenki College of Engineering and Technology (ECET), Hyderabad, India. He has received M.Tech. (Computer Science and Technology) from Andhra University, Visakhapatnam, India. Presently, he is a Research Scholar of JNTUH Hyderabad. He has been published and presented 34 Research and Technical papers in International Journals , International Conferences and National Conferences. His main research interests are Software Engineering, Testing, Software Metrics, and Software Quality.

Hymavathi. Bhadriraju is working as a Lecturer , Department of Computer Science and Engineering at Bharth University, Chennai, India. She has received M.Tech (CSE) from Bharth University, Chennai.,India. She has presented number of Technical papers in National Conferences. Her research interests are Software Engineering, Software Testing, Computer Networks, Network Security and Programming Languages.

K.Vikram is working as a Professor and Head, Department of Computer Science and Engineering at City Womens College of Engineering and Technology ,Hyderabad, India. He has received M.E from Anna University, Chennai,India. Presently he is a Research Scholar in JNTUH Hyderabad. He has published and presented good number of Technical and Research Papers in National and International Conferences. His research Interests are Software Testing, Image Processing , Computer Organization and Information Security.

Nazeer.Shaik is working as an Assistant Professor , Department of Computer Science and Engineering at Moghal College of Engineering and Technology (MCET), Hyderabad, India. He has received M.Tech (CSE) from Bharth University, Chennai, India. He has presented number of Technical papers in National Conference. His research interests are Software Engineering, Software Project Management, Computer Networks and Mobile Computing.

S.V. Achuta Rao is working as a Professor and Head, Department of CSE and IT at DJR Institute of Engineering and Technology (DJRIET), Vijayawada, India. He has received M.Tech. (Computer Science and Engineering) from JNTU, Kakinada, India. Presently, he is a Research Scholar of Rayalaseema University (RU), Kurnool, India. He has been published and presented good number of Research and technical papers in International and National Conferences. His main research interests are Data Mining, Networking, Image Processing, Software Engineering and Software Metrics.