A METRICS-BASED APPROACH TO THE TESTING PROCESS AND TESTABILITY OF OBJECT-ORIENTED SOFTWARE SYSTEMS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

TOLGA YURGA

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN
THE DEPARTMENT OF INFORMATION SYSTEMS

FEBRUARY 2009

Approval of the Graduate School of Informatics

Prof. Dr. Nazife BAYKAL
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Doctor of Philosophy.

Prof. Dr. Yasemin YARDIMCI
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Doctor of Philosophy.

Prof. Dr. Semih BİLGEN Co-Supervisor	Assoc. Prof. Ali H. DOĞRU Supervisor

Examining Committee Members

Dr. Ali ARİFOĞLU	(METU, II)	
Assoc. Prof. Ali H. DOĞRU	(METU, CENG)	
Prof. Dr. Semih BİLGEN	(METU, EEE)	
Assist. Prof. Dr. Aysu Betin CAN	(METU, II)	
Dr. Sadık EŞMELİOĞLU	(BİLGİ GRUBU)	

ABSTRACT

A METRICS-BASED APPROACH TO

THE TESTING PROCESS AND TESTABILITY OF

OBJECT-ORIENTED SOFTWARE SYSTEMS

Yurga, Tolga

Ph.D., Department of Information Systems

Supervisor: Assoc. Prof. Dr. Ali Hikmet DOĞRU

Co-Supervisor: Prof. Dr. Semih BİLGEN

February 2009, 195 pages

This dissertation investigates the factors that affect testability and testing cost of object-oriented software systems. Developing a software program which eases the testing process by increasing testability is crucial. Also, to assess whether or not the testing effort and cost consumed or planned is adequate or not is another critical matter this dissertation aims to answer by composing a new way to evaluate the links between software design parameters and testing effort via source-based metrics. An automated metric plug-in is used as the primary tool for obtaining the metric measurements. Our study is based on the investigation of many open-source projects written in Java to achieve our goals. By the help of the statistical evaluation of project data, we both propose a new model to assess testing effort and testability, and find significant relations and associations between software design and testing effort and testability of object-oriented software systems via source-based metrics.

Keywords: Software, Testability, Testing, Metric, Testcase

ÖZ

NESNE-YÖNELİMLİ YAZILIM SİSTEMLERİNİN

TEST SÜRECİNE VE TESTEDİLEBİLİRLİĞİNE

METRİK TABANLI BİR YAKLAŞIM

Yurga, Tolga

Doktora, Bilişim Sistemleri Bölumu

Tez Yoneticisi: Doç. Dr. Ali Hikmet DOĞRU

Ortak Tez Yoneticisi: Prof. Dr. Semih BİLGEN

Şubat 2009, 195 sayfa

Bu tez, nesne-yönelimli yazılım sistemlerinin testedilebilirliklerini ve test maliyetini etkileyen faktörleri araştırmayı hedeflemektedir. Testedilebilirliği arttırarak test sürecini kolaylaştıran bir yazılım geliştirmek çok önemlidir. Ayrıca, kaynak kod temelli metrikler yoluyla, tasarım parametreleri ve test gayreti arasındaki bağları değerlendirip yeni bir model oluşturarak, harcanması planlanan ya da harcanmış olan test gayretinin uygunluğunun değerlendirilmesi, bu tezin diğer bir önemli hedefini oluşturmaktadır. Metrik ölçümlemeleri için ana araç olarak otomatikleştirilmiş bir metrik eklentisi kullanılmıştır. Çalışmamız, hedeflerimize ulaşmak için Java dili ile yazılmış birçok açık-kaynak kodlu projenin detaylı araştırmasına dayanmaktadır. Proje verilerinin istatistiksel olarak değerlendirilmesi sayesinde, kaynak-kod bazlı metrikler üzerinden, hem test gayretini ve testedilebilirliğini değerlendirebilmek için bir yeni model öneriyoruz, hem de yazılım tasarımı ve test gayreti ve yazılımın testedilebilirliği arasında kayda değer ilişki ve bağlantıları ortaya koyuyoruz.

Anahtar Kelimeler: Yazılım, testedilebilirlik, test süreci, metrik, test vakası

To Didem

ACKNOWLEDGMENTS

I express sincere appreciation to Assoc. Prof. Dr. Ali Doğru for his guidance and insight throughout the research. I am grateful to the other supervisory committee members, Prof. Dr. Semih Bilgen and Dr. Sadık Eşmelioğlu, for their supports, suggestions and comments. To my wife, Didem and to both of my parents, I present sincere thanks for their unshakable faith in me and their willingness to endure with me since the beginning of my doctoral program and dissertation. I could not reach the end without them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
AST	abstract syntax tree
BMP	Bean Managed Entity Beans
BRMS	Business Rule Management System
CA	Afferent Coupling
CBO	Coupling Between Objects
CE	Efferent Coupling
CMMI	Capability Maturity Model Integration
CLS	Class
COTS	Commercial Off-The-Shelf
CVS	Concurrent Versions System
CW	Component Weightings
DIT	Depth of Inheritance Tree
DFT	Design-For-Testability
dLOC_CLS_NEW	Lines Of Code for Test Class - New Metric
dLOC_PKG	Lines Of Test Code for Package
dLOC_PRJ	Lines Of Code for Project
dLOCC	Lines Of Code for Test Class
dNOTC	Number of Test Cases for Test Class
dNOTC_CLS_NEW	Number of Test Cases for Test Class
dNOTC_PKG	Number of Test Cases for Package
dNOTC_PRJ	Number of Test Cases for Project
EAI	Enterprise Application Integration
EE	Enterprise Edition
EJB	Enterprise Java Bean
FCM	Factor-Criteria-Metrics
FOUT	Fan Out
GQM/MEDEA	Goal Question Metric / MEtric DEfinition Approach
HTML	HyperText Markup Language
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization

IT	Information Technology
J2EE	Java 2 Platform - Enterprise Edition
JCA	Java Enterprise Edition - Connector Architecture
JCR	Content Repository API for Java
JMS	Java Message Service
JMX	Java Management Extensions
JNDI	Java Naming and Directory Interface
JSF	Java Server Faces
JSP	Java Server Pages
JSR	Java Specification Requests
JTA	Java Transaction API
KLOC	Thousands Lines of Code
LCOM	Lack of Cohesion of Methods
LOC_CLS	Lines Of Code for Class
LOC_CLS_NEW	Lines Of Code for Class - New Metric
LOC_PKG	Lines Of Code for Package
LOCC	Lines Of Code for Class
MLOC	Method Lines of Code
MOP	Meta-Object-Protocol
MQMOOD	Metrics Based Quality Model for OO Design
MVC	Model View Controller
NASA	National Aeronautics and Space Administration
NBD	Nested Block Depth
NIO	New I/O : New Input/Output
NOC	Number of Classes
NOF	Number of Attributes
NOI	Number of Interfaces
NOM	Number of Methods
NOP	Number of Packages
NORM	Number of Overridden Methods
NOTC	Number of Test Cases (Number of JUnit Asserts)
NSC	Number of Children
NSF	Number of Static Attributes
NSM	Number of Static Methods
OJB	ObJect Relational Bridge
OO	Object-Oriented
PAR	Number of Parameters
PKG	Package
PL/SQL	Procedural Language/Structured Query Language
POM	Project Object Model
QMOOD	Quality Model for Object-Oriented Design
RDBMS	Relational Database Management System
REBOOT	REuse Based on Object Oriented Techniques
RFC	Response For Class

RMA	Richtmyer-Meshkov Abstractness
RMD	Normalized Distance
RMI	Richtmyer-Meshkov Instability
SATC	Software Assurance Technology Center
SIX	Specialization Index
SOA	Service Oriented Architecture
TCK	Technology Compatibility Kit
TCP/IP	Transfer Control Protocol / Internet Protocol
TLOC	Total Lines of Code
TNOF	Total Number Of Fields
TNOM	Total Number Of Methods
UDP/IP	User Datagram Protocol / Internet Protocol
VG	McCabe Cyclomatic Complexity
WMC	Weighted methods per Class
WS-BPEL	Web Services Business Process Execution Language
WSDL	Web Service Definition Language
XML	Extensible Markup Language

CHAPTER 1

INTRODUCTION

1.1 Problem Statement

Testing is an activity performed for evaluating product quality, and for improving it, by identifying defects and problems. Software testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior. [33]

Testing is an important software development activity as it consumes a significant amount of time and effort within an average software development project. It aims to determine whether a software program has errors. Testing is used to assess the compliance of a program to its intended specifications and to assess the reliability of the program to inputs that were not intended to be part of the specifications.

Testability of software emerges as an important attribute for software project management. Quality is the goal and there are so many facets of quality. Testing help the practitioners and the managers to assess the software being developed. Due to this well deserved emphasis, testability was chosen as one of the main problems to investigate in this research. The software industry suggests that it would create a better structuring in software development life cycle in case relations between testability and testing effort and significant design parameters are stated in mathematical models and promising ways, so that organizations may identify the areas to be more careful in software design, implementation and testing processes.

In general sense, it is cheaper to fix a defect the earlier it is found. It is natural and logical that, as the cost of a defect found in later stages of development cycle dramatically increase as it requires tracing and fixing more, earlier stages in the cycle.

For example, if a problem in requirements is found only post-release, then it would cost 10-100 times more to fix it comparing to the cost if the same fault was already found by the requirements review. [27]

The reason why testability is a crucial concept in software development lies in the effort testing consumes. According to available project data and measurements in the literature, software testing expends as much as 50% of development costs and comprises up to 50% of development time. Half of the project budget and effort goes to testing. Thus, to design a software in such a way that it would be easier to test and detect the defects would help decrease the effort and cost separated for testing process.

The literature review and interviews with the industry have revealed the fact that the crucial link between testability and testing effort and major design parameters is generally stated to be very important but its expression is generally stated informally rather than with a mathematical model or approach. The only important work to be mentioned was performed by Binder [2] points out the crucial link between major design parameters, i.e. encapsulation, inheritance, polymorphism and complexity and testability. This means that the key link can be stated as absent and vague, waiting for detailed exploration.

1.2 Purpose of This Study

In this dissertation, our primary concern is the factors that affect testability and testing effort of object-oriented software systems. The goal of this dissertation is to define our own model using the related software metrics. As a result of our study, we aim to identify relationships between software design, and testing effort and testability in object-oriented software systems via software metrics.

As suggested by the software industry, we aim the software companies to have a better structuring in software development life cycle as the relations between testability and testing effort and significant design parameters are stated in mathematical models and equations. Therefore, using our model and the results of our research, organizations may identify the areas to be more careful in software design, implementation and testing processes better than before.

1.3 Organization of the Dissertation

The organization of our dissertation is given below stating the contents of each chapter briefly.

Chapter 1 - Introduction : This chapter makes an introduction to our dissertation and states the problem we investigate, purpose the research aims and organization of our dissertation

Chapter 2 - Testability & Testing Effort : This chapter provides an overview of testability aspects and testing effort of software development. It gives information on the testability fish-bone concept driven by Binder, as this model helps us to understand and define testability with our vision and purpose.

Chapter 3 - Object-Oriented Design And Quality Models : This chapter provides an overview of quality factors and design parameters affecting testability and testing effort of object-oriented software development, as there are important relations among the most important design parameters and testability and testing effort to be discovered and examined in detail in our research. The most important quality models used for software design assessment will be analyzed to examine the importance of testability concept in these models and how we can use the design parameters to assess testing effort and testability concept.

Chapter 4 - Software Metrics : This chapter gives detailed information on the software metrics we have used in our research.

Chapter 5 - New Model On Testing Effort & Testability : This chapter summarizes the model we have composed as a result of our research.

Chapter 6 - Construction Of The Model : This chapter defines how we have composed our model. It begins by giving brief information on the projects used and continues by the details on the experimental framework and statistical methodology. Statistical results and their assessments are presented afterwards. Regression analysis performed to compose the equations of our model is stated finally.

Chapter 7 - Validation Of The Model : This chapter provides the details on how we have validated our model. It presents the results and assessments of the validation process, as well.

Chapter 8 - Discussions On The Model: This chapter provides our discussions on the model we have proposed.

Chapter 9 - Conclusions : This chapter provides the concluding remarks on our research. It summarizes the study, presents the contributions performed by our model and research and defines the future work to be performed.

CHAPTER 2

TESTABILITY & TESTING EFFORT

This chapter provides an overview of testability aspects and testing effort of software development. It gives information on the testability fish-bone concept driven by Binder, as this model helps us to understand and define testability with our vision and purpose..

2.1 Software Testing Process

IEEE defines software testing as "The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item."[16]

Testing consumes a significant amount of time and effort within an average software development project. There are different approaches to keep test costs under control and to increase the quality of the product under test [17]:

• improve the software specification and documentation,
• minimize or simplify functional requirements to ease testing,
• use better test techniques,
• use better test tools,
• improve the test process,
• train people and use qualified testers, and
• improve the software design and implementation.

Testing activity aims to detect the faults that may be present in a software program, before these faults may cause to the program to fail. A failure defines a condition when a program diverges from its requirements and produces a different output from the expected one.

The view of software testing has evolved towards a more constructive one. Testing is now seen as an activity, which starts at the beginning of software development life cycle and continues as the software lives. Before, it used to be seen as an activity, which starts after the implementation (coding) phase is complete, with the limited purpose of detecting faults existing in the software.

The main reason for this dramatic change lies in the fact that, it was successfully observed that preventing faults to occur help much more than detecting faults in later stages of the product lifecycle. Planning for testing now starts with the early stages of requirement process and test plans and procedures are systematically and continuously developed and refined, as development proceeds.

It is currently considered that the right attitude towards quality is one of prevention: it is obviously much better to avoid problems than to correct them. Testing must be seen, then, primarily as a means for checking not only whether the prevention has been effective, but also for identifying faults in those cases where, for some reason, it has not been effective. It is perhaps obvious but worth recognizing that, even after successful completion of an extensive testing effort, the software could still contain faults. The remedy for software failures experienced after delivery is provided by corrective maintenance actions, which also means testing process continues in the maintenance phase, as well.

Software testing occurs during multiple phases of the construction of a software system. Typically the software development methodology determines both the kind of testing, and the phase(s) during which testing is done. Since methodology is not our focus here, it will be enough to briefly describe the different kinds of testing that are common in practice. It is useful to consider the several aspects of testing separately.

A common practice of software testing is performed by an independent group of testers after the functionality is developed before it is shipped to the customer. This practice often results in the testing phase being used as project buffer to compensate for project delays, thereby compromising the time devoted to testing. Another practice is to start software testing at the same moment the project starts and it is a continuous process until the project finishes. [7]

In counterpoint, some emerging software disciplines such as extreme programming and the agile software development movement, adhere to a "test-driven software development" model. In this process unit tests are written first, by the software engineers (often with pair programming in the extreme programming methodology). Of course these tests fail initially, as they are expected to. Then as code is written it passes incrementally larger portions of the test suites. The test suites are continuously updated as new failure conditions are discovered, and they are integrated with any regression tests that are developed. Unit tests are maintained along with the rest of the software source code and generally integrated into the build process (with inherently interactive tests being relegated to a partially manual build acceptance process).[39]

The following overview of software testing is based on the Software Engineering Body of Knowledge (SWEBOK). [33] Testing can be done on the following levels [39]:

- **Unit testing** tests the minimal software component, or module. Each unit (basic component) of the software is tested to verify that the detailed design for the unit has been correctly implemented. In an object-oriented environment, this is usually at the class level, and the minimal unit tests include the constructors and destructors.
- **Integration testing** exposes defects in the interfaces and interaction between integrated components (modules). Progressively larger groups of tested software components corresponding to elements of the architectural design are integrated and tested until the software works as a system.
- **System testing** tests a completely integrated system to verify that it meets its requirements. On the one hand, the system can be validated against the non-functional requirements, such as performance, security, reliability or interactions with external systems. On the other hand, the functionality implemented by the system can be compared to its specification.

Testing can have several objectives. Although, the base objective of testing is verification of the implemented source code to the specifications, however, the reference to be used for verification can be different.

- **Acceptance/qualification testing** is done to verify that the system implements the customer's requirements correctly. Usually the testing is done by (future) users of the system. In addition to verifying whether the required functionality is present in the system, (future) users are also likely to be concerned about the user-interface and performance characteristics.
- **Installation testing** aims to verify the software upon installation in the target environment.
- **Alpha and beta testing** are performed before shipping the final version of software. The software is delivered to a small, representative set of potential users for trial use.

- **Alpha testing** is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing, before the software goes to beta testing.
- **Beta testing** comes after alpha testing. Versions of the software, known as beta versions, are released to a limited audience outside of the programming team. The software is released to groups of people so that further testing can ensure the product has few faults or bugs. Sometimes, beta versions are made available to the open public to increase the feedback field to a maximal number of future users.

- **Conformance testing/Functional testing/Correctness testing** is done to determine if the system has correctly implemented the specification of functionality. Typically, a team separate to the development or maintenance teams would perform this task.
- **Reliability achievement and evaluation** is sometimes done by executing test cases obtained from a typical operational profile for the system. The rate of failure observed during such a test session can then be used to derive statistical measures of the reliability of the system.
- **Regression testing** is performed to make sure that a modification of a certain part of the system has not inadvertently broken other parts of the system. After modifying software, either for a change in functionality or to fix defects, a regression test re-runs previously passing tests on the modified software to ensure that the modifications have not unintentionally caused a *regression* of previous functionality. Regression testing can be performed at any or all of the above test levels. These regression tests are often automated.
- **Performance testing** specifically aims to verify that the software meets the specified performance requirements, for instance, capacity and response time.
- **Stress testing** exercises software at the maximum design load, as well as beyond it.
- **Back-to-back testing** aims to compare different implemented versions of a software product. A single test set is performed and the results are compared.
- **Recovery testing** aims to verify software restart capabilities after a “disaster.”
- **Configuration testing** aims to compare a software under different configurations built to serve different users.
- **Usability testing** aims to evaluate how user-friendly the software product is. To be

able to define the software's usability level, its documentation, its ability to recover from errors, its easiness to be used and learned by the end-users are all assessed.

- **Test-driven development** promotes the use of tests as a surrogate for a requirements specification document rather than as an independent check that the software has correctly implemented the requirements.

IEEE defines a **test case** as: "Documentation specifying inputs, predicted results, and a set of execution conditions for a test item" [16] The selection of test cases plays an important role in software testing process. We will now discuss the ways in which test cases can be selected.

There are many forms of test techniques stated in the SWEBOK [33]. It is difficult to define a basis for classifying all these techniques. A general classification divides test techniques into 2 classes:

- White-box testing (also called *glassbox testing*),
- Black-box testing

White-box testing defines the group of test techniques in which the tests rely on information about how the software has been designed or coded. White-box testing refers to the creation of test cases by exploiting knowledge of the implementation (i.e. the source code) of the system under test. Therefore, white-box techniques are typically applied by the same developers that wrote the code.

Several aspects of the source code can be targeted by white-box techniques. For example, possible techniques are based on the control-flow, data-flow or call behavior of the code being tested. Observing the effects of modifications made to certain parts of the code, so-called mutation analysis can also be classified as a white-box technique. [7]

Black-box testing is the opposite of white-box testing, in the sense that no knowledge of the implementation is used to generate test cases. Black-box testing defines the group of test techniques in which the tests rely only on the input/output behavior. This approach enables people without knowledge of the internals of a system to apply these techniques. [7]

Many black-box techniques take the specification of the system as a starting point. The specification should provide information about the domains of inputs and outputs of the system, and describe the implemented functionality. Using this information, the tester should be able to generate input/output pairs that represent correct executions of the system. In other words, for every pair, the system should result in the specified output value when given the specified input value. Clearly, one such pair exactly represents a test case.

In general, a system that would pass all possible test cases implements its specification correctly. However, exhaustively testing a system is not a feasible practice, since most interesting systems will likely involve input and output domains, which are very large. Therefore, a number of techniques exist to reduce this problem. These provide ways to select a set of test cases that will provide a reasonable level of confidence in the correctness of a system that passes the tests. Examples of these techniques are partitioning of the domains in equivalence classes, boundary-value analysis, random testing, and statistical testing based on an operational profile.[7]

2.2 Testability

IEEE defines testability as "(1) 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. (2) The degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met." [16]

Binder [2] defines the testability of a software system as the relative ease and expense of revealing software faults. In other words, the testability of a software system is a direct indicative of the amount of effort, i.e. ease, and cost, i.e. expense needed to test the system.

Binder's testability fishbone (Figure 1) shown below is the major starting point in composing our point of view on software testability. This diagram indicates that, the amount of effort, both labor and cost, that one should expect to spend on testing process, given a certain desired degree of validity, is therefore a result of properties of both the process and the software. [2]

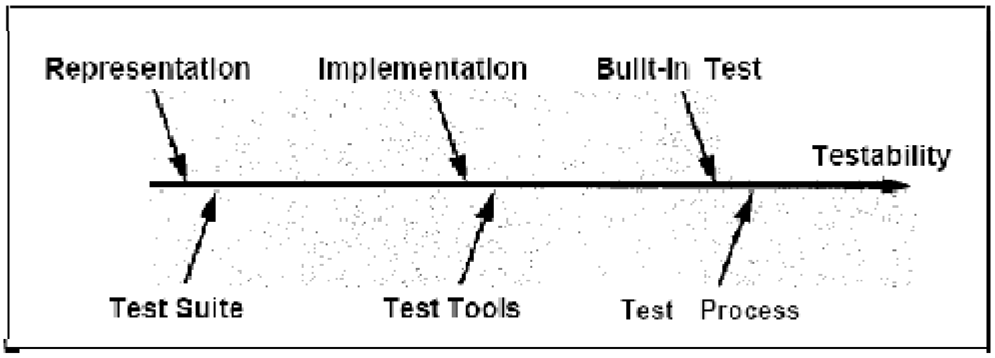


Figure 1 : Testability Major Fishbone by Binder [2]

The diagram below (Figure 2 [7]) shows a simplified version of the original one by Binder. In the diagram, the “bones” of the figure indicate the important aspects of software development lifecycle with respect to the testing process, i.e. effort. Binder states that testability is as much a process issue as it is a technical problem and that there are six primary testability factors, each of which may facilitate or hinder testing in many ways. These six primary testability factors are:

- Representation,
- Implementation,
- Built-in Test,
- Test Suite,
- Process Capability, and
- Test Tools.

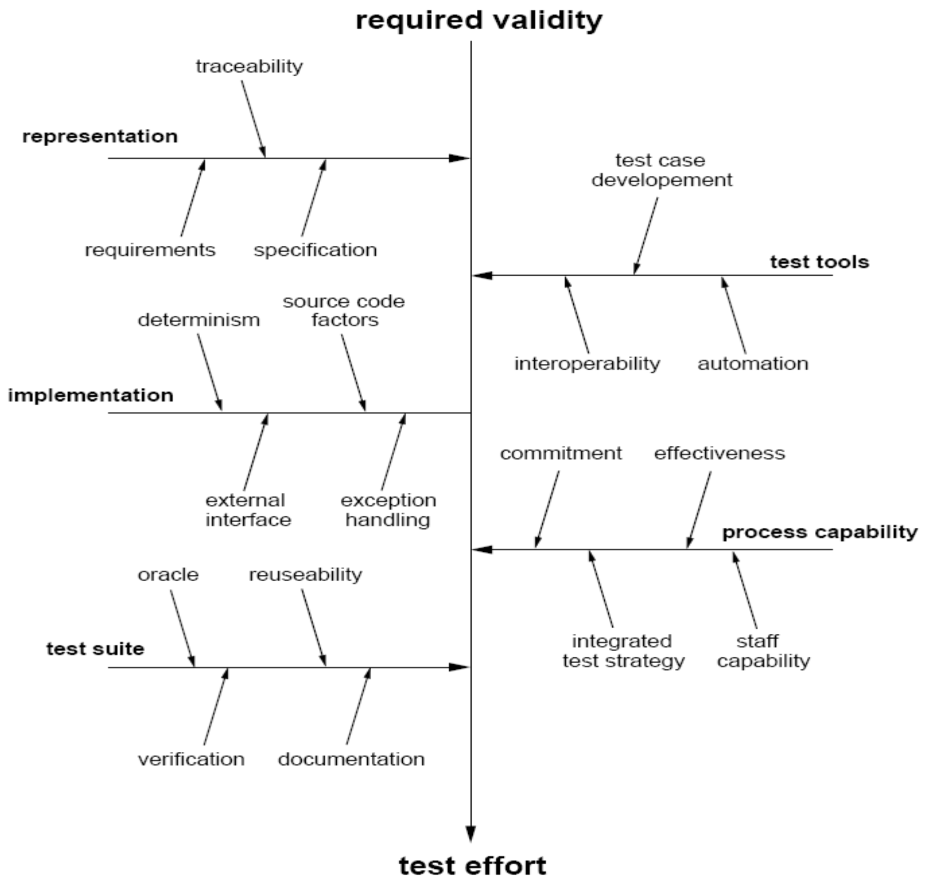


Figure 2 : Simplified Testability Fishbone by Bruntink [7]

As seen on the figure, the main testing goal of the software project is set by the required degree of validity. It is a natural consequence that the expected testing effort increases as the goal, the required degree of validity, is requested at higher levels. Representation bone deals with requirements, specification, traceability and separation of concerns issues, as a usable representation is necessary to develop test cases. As implementation characteristics determine controllability and observability, this bone takes care of main coding issues, such as exception handling and interfaces with external systems.[7]

Built-in Test capability provides explicit separation of test and software functionality, as systematic addition of the members of the bone, i.e. set/reset, reporters, and assertions to a class helps to improve controllability and observability. Test suite bone deals with quality aspects of the members of the test suite, the test cases and plans to use them. Test tools bone takes into account the automated test tools that help the testing process of the software project. The final bone, process capability focuses on the overall process capability and maturity as the deficiency of effective organizational approach to testing and its antecedents makes the whole testing process irrelevant and unnecessary.

2.3 Design for Testability

The concerns regarding the cost-effectiveness of OO testing have created a new concept: Design-for-testability (DFT). It focuses on early life-cycle activities that can increase the testability of systems with a primary aim to increase both the ease and value of testing such that the benefits of object-oriented design and development are fully realized.

Binder states that "Design for testability is a strategy to align the development process so that testing is maximally effective under either a reliability-driven or resource-limited regime."

2.4 The Testability Fish-Bone

The focus of this dissertation is primarily based on the testability fish-bone concept introduced by Binder. Therefore, we will try to expand the information given above and examine the fishbone in detail. As shown on the fishbone diagram below, software testability is stated to be a result of six factors:

- Characteristics of the representation
- Characteristics of the implementation
- Built-in test capabilities
- The test suite (test cases and associated information)

- The test support environment
- The software process to conduct testing process in

2.4.1 Degree of Validity and Testing Effort

The major input and output information on the spine of the fishbone (Figure 2) are the required degree of validity and the required testing effort, consecutively. The fishbone contains major and minor bones. The minor bones compose the major bones and the major bones all together form the fishbone.

The required degree of validity defines the level a software project is to be tested. The higher the degree of validity, the higher testing effort is needed, as software that is required to have a high degree of validity will need to be tested thoroughly before it can be claimed the requirement is met.

The required degree of validity varies according to the software project's development purposes and the adhered audience. For example, an embedded software system to be run on a military airplane is a safety-critical system and expected to run with minimum or no error, as safety-critical systems are often required to meet very strict validity requirements; maximally allowable failure rates are typically stated explicitly. On the other hand, a COTS application will not be expected to have the same degree of validity of a safety-critical system.

A software project may have a defined degree of validity or not. If the project has a predefined degree, the testing effort needed is a result of the software development stages and their related aspects, as the goal of the testing is already at the hand. It is not right to expect a required degree of validity at all times, depending on the context and nature of the project. In such cases, it is not straightforward to define the testing effort needed. It will either be according to the available testing effort the software project owner is willing to spend on testing process, or to the defined testing methodology defined in software development process of the project owner.

In common, when a required degree of validity is not defined, the testing effort may be performed depending on some other criterion, which indicates whether necessary testing has been done. An example to such testing criterion is code coverage criterion, common in the context of white box testing, in which the tests rely on information about how the software has been designed or coded. This criterion indicates the extent to which a certain aspect of the code has been "covered" by testing.

In many web-based Java projects, in case of undetermined degree of validity, a certain level of code coverage criterion is defined. For example, the project may be expected to satisfy a minimum of 70% code coverage ratio, defined by the project manager or testing manager. An upper bound is also necessary, as the defined code coverage ratio may require more effort than available resources. Thus, the maximum number of test cases to be generated may be defined as well, to define an upper bound on the testing effort. This means there is a certain trade-off at the testing process due to minimum and maximum constraint. The testing team will probably have to pay more attention to more critical parts of the software. This raises an important question. Which part of the software do you have to pay more attention? We will try to examine and answer this question in the following chapters of our dissertation, as the testing effort is valuable and you have to use your worthy resources in the most effective and efficient manner .

2.4.2 Representation

Representation major bone consists of the following minor bones:

- Requirements
- Specification
- Traceability
- Separation of Concerns

Requirements are the key components to validation as they capture the expectations of customers in written form. They are a crucial source of test cases and plans to assess whether the implementation is complete and correct. IEEE/ASI standard 830 defines a good requirement to have the following desired aspects; Unambiguous, verifiable, complete, consistent, feasible, traceable, modifiable, useful for maintenance.

A specification describes the architectural and structural design composed according to the obtained requirements to provide input to the implementation phase. A suitable form of design document should include detailed information about the output of the design process, including the organization of software components, dependencies, interfaces, and detail of algorithms and data structures. It must be complete to cover all aspects of the system as the implementation stage needs precise inputs to compose the output, without any need for further examination and determination.

Traceability is crucial during testing process, as it is important to trace relationships between a given specification and a given software component, and also between a given specification and a given requirement. Using traceability matrices and diagrams, it is easier to develop complete and accurate test plans for any scale of software systems and to trace whether or not software is correctly verified and validated by the testing process. Configuration management helps the specifications to be current, meaning having up-to-date test plans. Separation of Concerns is a key software engineering principle, which aims to divide large components into smaller components to increase controllability and observability.

2.4.3 Implementation

Implementation major bone consists of the following minor bones:

- Structure
- Fault Sensitivity
- External Interface
- Determinism
- Exception Handling

In a software project, all testing activity is performed on the source code, i.e. implementation of the project. The implementation may be seen as the mirror of all activities and work done prior to this stage. Requirements obtained from the customer are documented in a clear way to enable engineers to design the architectural and structural framework of the software to be coded. Thus, the source code is a crucial output of any software project.

Structural factors of the source code are the major focus fields in this dissertation, as we use metrics to assess source code. They will be explained in more detail in the following chapters together with the related source code metrics.

Fault sensitivity is the probability that a fault will be revealed by a randomly selected test case, given that a fault is indeed present. Fault sensitivity is directly related to testability, meaning low sensitivity corresponds with low testability. Testability encompasses the whole program and its sensitivities under a given input distribution. Sensitivity analysis is the process of determining the sensitivity of a location in a program. [36]

External interface complicates testing and thus testability, the external relationships decrease controllability and observability. Determinism is another implementation factor, meaning the extent to which the tested class or software component does not require asynchronous cooperation with other tasks. High determinism provides repeatability, as repeating the test as different times after major changes or build is a desired need.

Handling exceptions thrown in the source code is vital, as unhandled cases may cause the software to fail. Thus, to be able to test *exception handling,* consistent usage of language-supported features and a related design strategy is required.

2.4.4 Built-in Test

Built-in Test major bone consists of the following minor bones:

- Driver
- Set/Reset
- Safety
- Reporter
- Assertions

Built-in test capability provides explicit separation of test and application functionality. The systematic addition of set/reset, reporters, and assertions to a class is a simple way to provide effective control and observation. Attempts to approximate BIT with application methods is a partial solution at best. If a standard test interface is included in all classes, additional development overhead is minimal and the potential payback is great.[2]

A *driver* is a special-purpose class that activates the class under test. IEEE defines a test driver as "*A software module used to invoke a module under test and, often, provide test inputs, control and monitor execution, and report test results*."[16] *Set/Reset* provides effective state-based testing. A set/reset method helps an object to be set to a predefined value, independent of its current state. A *safety* provision is advisable to prevent inadvertent or willful misuse of built-in test services. A *reporter* concrete state of an object, i.e. its private internal state. The reporter must provide complete reporting of the abstract state in case the class under test does not give the necessary details. Assertions are special code segments, described by IEEE as "*A logical expression specifying a program state that must exist or a set of conditions that program variables must satisfy at a particular point during program execution.*"

2.4.5 Test Suite

Test Suite major bone consists of the following minor bones:

- Oracle
- Reusable
- Verified
- Documentation

Test suite bone deals with quality aspects of the members of the test suite, the test cases and plans to use them. A test suite is a collection of test cases and plans to use them. Therefore, the aspects of a test suite itself are major factors to determine the testing effort.

The generated test cases and scenarios should be suitable for automated tool execution. An oracle is a mechanism for producing expected results, which are necessary for test cases. A useful oracle must be feasible, otherwise testing would be impossible. In addition, an oracle should be based on the specifications, as the architectural and structural design composed according to the obtained to ease traceability, as well. A useful oracle must be efficient, otherwise it would require more testing effort.

A test suite should be reusable as it provides economic benefit to use an existing utility, which increases the total amount of testing and thus testability. To be able to create reusable test suites, it is inevitable to use necessary tools to ease configuration management control and traceability. By this way, the test team may perform test cases on different versions of the product line.

Documentation is crucial for test suites. Test suites need necessary documentation about relevant details on tests to be performed, test plans, test cases, test design, test procedures and test history. A good test suite needs to be verified, as it may also contain errors. In case of such faulty test suites, unexpected results may happen, an implementation that must be rejected may be accepted or a correct implementation that must be accepted may be rejected.

2.4.6 Test Tools

Testing without automated tools means accepting either to test less or to test the same implementation with more effort and cost. In both ways, you must sacrifice testability.

The testing environment (bed) is important, as it needs functions to initialize a system and its environment, execute test scripts, and replay scripts under predefined conditions. Recently, many commercial and open-source solutions are available for object-oriented programming practices. Definition of test cases is much easier in the presence of test tools, as many generators are available to compose both test cases and related data to be used in the tests.

Interoperability is another important issue. As we pay crucial attention to automation, making different automated tools to work together helps need for effort to pass data among these systems.

2.4.7 Process Capability

A process may be defined as the collective work whose participants come together to support a certain activity, including organizational structure, human and other types of resources. The process has a great influence on the testing process, directly and indirectly. It is crucial to have management to support to enable and effectively run testing process.

Without any of the participants of the process, the process itself may not operate or operate inaccurately. In case you do not have necessary resources for testing, it is unnecessary to have the perfect analysis and design as you do not have time and labor to test these stages before you deliver the final product to the customer.

The staff should be well-trained, motivated and experienced to fulfill a successful testing process. The effectiveness of the testing process is also related to importance it is given. The more you view the testing process as an essential and irreplaceable component of the software development process, the more effective the testing process becomes.

A chain is as strong as its weakest component. Seeing the whole software development process as a chain, it is no use to strengthen testing process without paying attention to other processes. In addition, to get the right output, one should submit the right input. The preceding processes must be revised to work in harmony with testing process, which is the major focus of "Test-Driven Development" by Kent Beck [1].

The testing order of the components under test should be compatible with the sequence used during development. For example, a software system developed in top-bottom fashion should be tested accordingly.

An integrated test strategy defines the contextual meaning of testing process. Vertical integration takes into account the relationships among the classes, clusters of classes, and application systems and testing process contains a well-defined among these relationships. Horizontal integration spreads testability concept into all stages of software development process, analysis and design (representation), coding (implementation), testing (test suite) and subsequent iterations of reuse and maintenance. Verification and validation integration requires taking into account other quality assurance practices, such as prototyping, inspections, and reviews for all stages and work products, and having a balanced and expected testing process to include those quality assurance practices.

CHAPTER 3

OBJECT-ORIENTED DESIGN AND QUALITY MODELS

This chapter provides an overview of quality factors and design parameters affecting testability and testing effort of object-oriented software development, as there are important relations among the most important design parameters and testability and testing effort to be discover and examined in detail in our research. The most important quality models used for software design assessment will be analyzed to examine the importance of testability concept in these models and how we can use the design parameters to assess testing effort and testability concept.

3.1 Object-Oriented Programming

Due to the increasing complexity of software programs, a need for a new approach has become obvious, which resulted in the occurrence of Object-Oriented Programming. It was commonly used in mainstream software application development after the early 1990s. This approach provides us the necessary mechanisms to deal with this increasing complexity. Some of these mechanisms are specific to object-oriented programming, but some are not. In this section, we will discuss the major ones of these mechanisms.

IEEE [16] defines an object-oriented language as "A programming language that allows the user to express a program in terms of objects and messages between those objects."

Booch [4] defined object-oriented programming as: "Object-oriented programming is an implementation method in which programs are organized in object collections that cooperate among themselves, each object representing an instance of a class; each class is part of a class hierarchy and all classes are related through their inheritance relationships."

Sommerville [34] defines object-oriented design as "Object-oriented design is a design strategy where system designers think in terms of 'things' instead of operations or functions. The executing system is made up of interacting objects that maintain their own local state and provide operations on that state information."

3.2 Quality Factors & Design Parameters

At the stage of clarifying the metrics to be used in my research, I have preferred the metrics on the most important design parameters. Detailed explanations on these parameters can be found in APPENDIX G.The preferred design parameters are explained belowe briefly. [13] These design parameters helped us to determine the most significant design metrics to be used in our model.

Table 1 : Design Parameters Preferred in our Study

Design Parameter	**Brief Description**
Coupling	IEEE [16] defines coupling as "The manner and degree of interdependence between software modules". In an object-oriented design, coupling refers to relationships and dependencies between the communicating modules.
Cohesion	IEEE [16] defines cohesion as "The manner and degree to which the tasks performed by a single software module are related to one another".
Complexity and Size	IEEE [16] defines complexity as "(1) The degree to which a system or component has a design or implementation that is difficult to understand and verify; **(2)** Pertaining to any of a set of structure-based metrics that measure the attribute in (1)".

Data Abstraction	IEEE defines [16] data abstraction as “(1) the process of extracting the essential characteristics of data by defining data types and their associated functional characteristics and disregarding representation details. (2) The result of the process in (1)”.
Modularity	IEEE defines [16] modularity as “The degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components”.
Encapsulation	IEEE defines [16] encapsulation as “A software development technique that consists of isolating a system function or a set of data and operations on those data within a module and providing precise specifications for the module”.
Inheritance	Budd defines inheritance as “the principle that knowledge of a more general category is also applicable to a more specific category”. [8]
Polymorphism	The term polymorphic has Greek roots and roughly means “many forms”, as “poly” means “many” and “morphos” means “form”.

3.3 Quality Models for Software Design Assessment

The main aims of the models to assess software design can be simply stated as "*to better quantify quality*". Many models have been developed so far by many researchers and practitioners. The instruments these models use to measure software product quality are set of quality attributes, characteristics and set of metrics. The common point in all model developers is that "internal product characteristics influences external product attributes" [41]. Below are given some quality metrics models currently popular in software industry.

3.3.1 The Factor-Criteria-Metrics Model

This model is generally adopted as a basis of software evaluation. In the late of 1970s, McCall [26] and Boehm [3] respectively proposed two software quality hierarchy models. The main principle of this model is that each attribute can be decomposed into a set of factors, which themselves can be decomposed into a set of criteria. Moreover, the criteria can be attained from a set of software measurements, which is also called software metrics [42]:

The McCall quality model is organized around three types of Quality Characteristics:

- Factors (To specify): They describe the external view of the software, as viewed by the users.
- Criteria (To build): They describe the internal view of the software, as seen by the developer.
- Metrics (To control): They are defined and used to provide a scale and method for measurement.

Karlsson lists the assessment hierarchy in his paper as follows [18]:

- Attribute. A high level goal concerning the product, not necessary an organizational goal, e.g., reusability.
- Factors are used at the customer and management level. All non-functional requirements for the software are stated at this level. This can be, e.g., "The software should be highly maintainable".
- Criteria are a set of requirements for each factor and are used at the software designer and project manager level. An example for a factor is: "To make the software highly maintainable we must make the software consistent and self-descriptive."
- Metrics is software-related measurement, to determine the criteria. They are used at the software and document level. If the criteria example is continued then the

metrics can be “To make the software self-descriptive we must provide it with a header that describes its functionality and parameters”. The metrics is then computed by checklists or by counting the software characteristics.

3.3.2 ISO 9126 Model

This model is an international standard for the evaluation of software quality, which is a derivation of McCall's model. It defines software quality as *"The totality of features and characteristic of a software product that bear on its ability to satisfy stated or implied needs"*. The standard is divided into four parts, which address, respectively, the following subjects: quality model; external metrics; internal metrics; and quality in use metrics.

The standard claims that quality is composed of 6 factors: functionality, reliability, efficiency, usability, maintainability, portability, and that one or more of them are enough to describe any component of software quality.

- **Functionality** - *A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.*
    - Suitability
    - Accuracy
    - Interoperability
    - Compliance
    - Security
- **Reliability** - *A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.*
    - Maturity
    - Recoverability
    - Fault Tolerance
- **Usability** - *A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users.*
    - Learnability
    - Understandability
    - Operability
- **Efficiency** - *A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.*

    - Time Behavior
    - Resource Behavior
- **Maintainability** - *A set of attributes that bear on the effort needed to make specified modifications.*
    - Stability
    - Analyzability
    - Changeability
    - Testability
- **Portability** - *A set of attributes that bear on the ability of software to be transferred from one environment to another.*
    - Installability
    - Replaceability
    - Adaptability
    - Conformance

This model does not provide proper definition of the lower-level details and metrics needed to attain a quantitative assessment of product quality, which is stated to be its most important deficiency.

3.3.3 REBOOT (REuse Based on Object Oriented Techniques) Quality and Reusability Models

The objective of the REBOOT Project was set to enhance productivity and quality in software development by promoting and assisting reuse. The goal of the REBOOT project was to provide:

- a model for the description of a reusable component using an entity-relationship schema,
- a model for measuring the quality and reusability of a component,
- a model for measuring the costs and benefits of reuse,
- a methodology for software development for and with reuse,
- a complete training package,
- an industrial environment for supporting reuse,
- a database of general-purpose reusable software components,
- several databases of domain-specific, reusable components, and
- a study of the managerial, commercial and legal aspects of reuse.

This project proposed a general quality model and a general reusability model based on FCM model. Their main objective was to arrive at a reasonable computation of quality and reusability. They distributed questionnaire to software engineers in five European countries and then adopted the requirements the software engineers considered important for a component as factors in their FCM model.

All the factors are cost-related, productivity-related, or probability-related. With the starting point of decomposing an activity, into a subset of activities, a set of criteria of this factor are then defined. Metrics into which each criterion is decomposed are obtained from the literature, the questionnaires and through discussions with application projects. They may be computed from two sources: answers to checklist questions or counting software characteristics. [42] The model decomposes reusability into four management-related factors and decomposes quality to two factors, each of which is then decomposed to a set of more detailed software-related criteria.

3.3.4 Dromey's Quality Model

Geoff Dromey [11] states that quality characteristics or high-level attributes, cannot be built directly into software, but instead important product properties, like modules without side effects, can be identified built and measured as tangible properties, influencing or inducing high-level attributes, such as reliability or maintainability. These are intangible properties in the sense that they cannot be directly measured.

In order to point out this influence, product properties must be linked with high-level attributes. For this, a quality model framework is proposed. The important thing is to focus on those high-level attributes that describe the priority requirements for the software. Products are built of components. Rules-of-form govern each component type. So, product quality is determined by the choice of the components, tangible properties of individual components, tangible properties of the component composition.

Rules-of-composition govern the way components are used in the context of other components. Violating some of these rules may affect the functionality of the system or even non-functional requirements, such as performance. [22]

Component selection, component property identification, and component composition determine overall quality altogether. Since software quality is often discussed in terms of high-level attributes such as functionality, reliability, etc., a set of complete, compatible, and non-overlapping high-level quality attributes needs to be identified. Links need to be established between tangible product properties and the intangible quality attributes. Each link established should empirically be verified for each product property.

Dromey proposes the following five steps in constructing a testable, assessable and refinable product quality model: [42]

- Identify a set of high-level quality attributes for the product.
- Identify the product components.
- Identify and classify the most significant, tangible, quality-carrying properties for each component.
- Propose a set of axioms for linking product properties to quality attributes.
- Evaluate the model, identify its weaknesses, and either refine it or scrap it and start again.

3.3.5 QMOOD (Quality Model for Object-Oriented Design)

QMOOD (Quality Model for Object-Oriented Designs) was proposed as a hierarchical model used to assess object-oriented design quality. In this model, a hierarchy of levels is used to relate high-level and difficult-to-assess quality attributes to the low level of details.

QMOOD is a quality model for assessing high-level external quality attributes such as reusability, functionality, and flexibility of object-oriented designs based on the internal properties of design components.

In this model, tangible design properties (both structural and functional) of object-oriented design components such as classes, are used to generate object-oriented design metrics, which evaluate the extent of the tangible properties in the design components. The tangible design properties of components and their manifestation in a product contribute to object-oriented design properties, which are high-level software properties (not directly tangible) such as abstraction, encapsulation, coupling, and cohesion.

The model relates object-oriented design properties to a set of high-level external quality attributes using empirical and anecdotal information. The relationship, or links, from design characteristics to external quality attributes are assigned values based on the importance of their contribution to a particular quality attribute. The model is validated by using empirical and expert opinion to compare with the model results from several large commercial object-oriented systems [42]

The model can be easily modified to include different suites of design metrics, design properties, linking relationships, and quality attributes, thus providing a practical object-oriented design quality assessment model adaptable to a variety of demands. Figure below shows the four stages (levels L1 through L4) and three mappings (links: L12, L23, and L34) used (to connect the adjacent levels), in QMOOD**.** The methodology, by which this model is developed, is a specific extension of Dromey's generic quality model methodology.

The levels in the model may be summarized as follows: [42]

- ***Identifying quality attributes (L1)*** : QMOOD uses a new set of using a new set of six quality attributes based on the six attributes of ISO 9126, which are "*reusability"*, "*flexibility*", " *understandability* ", "*functionality*", "*extendibility*", "*effectiveness*".
- ***Identifying Object-Oriented Design components and their quality-carrying properties (L4)*** : Design components proposed in QMOOD include objects, classes, relations between the objects and classes of a design, and attributes and methods of a class that can be considered as low level design components, all of which can be easily presented in object-oriented design and programming language. Other high-level components that are identifiable and define the architecture of an object-oriented design are clusters, patterns, and framework.
- ***Identifying fundamental design properties that reflect quality characteristics of Object-Oriented components (L2)*** : QMOOD proposes twelve design properties, which include design size, abstraction, encapsulation, modularity, coupling, cohesion, complexity, messaging, composition, inheritance, polymorphism, and class hierarchies. While the former seven properties are frequently used as being representative of design quality characteristics in both structural as well as object-oriented development, the latter five properties represent new design concepts, which have been introduced by the object-oriented paradigm, and are thus vital to the quality of an object-oriented design.
- ***Relating Component properties to design properties :*** The set of quality-carrying properties of fundamental components (attributes, methods and classes) is large but highly overlapping. Most of them can all be classified into the smaller set of twelve fundamental design properties.
- ***Defining Object-Oriented metrics to assess design properties (L3, L23, L34):*** Each design property identified in the QMOOD model is sufficiently well defined to be objectively assessed by using one or more well-defined design metrics. All design metrics in QMOOD are based solely on information available during design time. Therefore the model defines a set of new object-oriented metrics that were solely based on class definitions, each of which has been classified as either being system measures or class measures.

- ***Relating and defining linkage weights from design properties to quality attributes (L12):*** The model relates design properties to quality attributes subjectively by considering the relations between each design property and quality attributes based on their experience and empirical knowledge of object-oriented systems. For example, using the coefficients in the "Quality Attributes - Design Property Weights" table of the model, "*Effectiveness*" may be found as below**:**

  Effectiveness = 0.2 * (Abstraction + Encapsulation + Composition + Inheritance + Polymorphism)

- ***Forming the model equation :*** The figure below summarizes model, using the computations stated in the above items to connect quality attributes to design metrics.

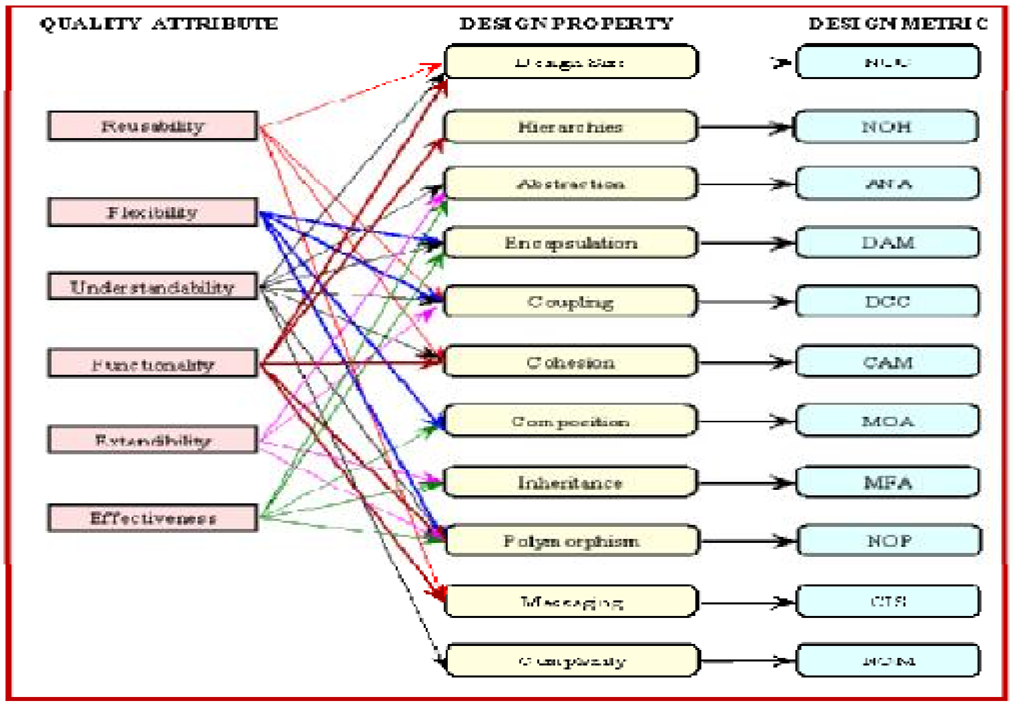


Figure 3 : Summary of QMOOD Model

3.3.6 MQMOOD (Metrics Based Quality Model for OO Design)

Another model was proposed, which extends Dromey's generic quality model to develop the improved Metrics Based Quality Model for Object Oriented Design (MQMOOD) [20] for the assessment of high-level design quality attributes in object oriented design. In this model, the design properties of classes, objects and their relationships have been evaluated using a suite of object oriented design metrics [21].

Proposed model relates design properties such as encapsulation, inheritance, coupling and cohesion to a set of high-level quality attributes such as efficiency, complexity, understandability, reusability and testability/maintainability identified by Software Assurance Technology Center (SATC). The relationship or links, from design properties to quality attributes are weighted in accordance with their anticipated influence and importance. The model seems to be useful as a practical quality assessment tool in design phase of the software development life cycle and may be adaptable to variety of demands. The new model deals with the three principal elements: product properties that influence quality, a set of high-level quality attributes, and a means of linking them. It extends Dromey's generic quality model shown in the figure above, which involves the following steps:

- Identification of product properties (Object Oriented Software) that influences quality.
- Selection of a set of high-level quality attributes (relevant of course to the stage under study).
- Identification of Object Oriented Design Metrics
- A means of linking of them.

The model gives the computation formula for testability as:

0.08 * *Encapsulation* + **1.12** * *Inheritance* + **0.97** * *Coupling*

3.4 Testability and Quality Models

The new trend [38] towards modifying the old quality factors of QMOOD with more recent, important and useful factors such as testability indicates that our study is on the right track. The quality models used to assess software design quality, especially QMOOD and MQMOOD are the two important and detailed models that have important effect on this study.

QMOOD gives a great insight to our study although it does not include the favorite quality factor "*testability*". The model can be easily modified to include different suites of design metrics, design properties, linking relationships, and quality attributes, thus providing a practical object-oriented design quality assessment model adaptable to a variety of demands. Thus, the model may be suitable to insert testability concept inside and explore the links between testability and the related design parameters.

MQMOOD is a modified version of the original QMOOD model. This model includes testability as one of the high-level quality factors and tries to observe the relationships among the four major design parameters, i.e. encapsulation, coupling, cohesion and inheritance. Although this new model combines testability and maintainability under one item only, it is successful to bring a new insight towards the relationships among design parameters and testability. [20]

CHAPTER 4

SOFTWARE METRICS

In this chapter, we describe the metrics in we have selected for our research in detail. These metrics are used in different level for our measurement, as they adhere to different levels of software components. Most of the metrics we will use depend on the ones proposed in Chidamber and Kemerer's article [9] and Binder's article [2]. According to the evaluation criteria, they can be classified into four groups:

- Method Level Metrics
- Class Level Metrics
- Package Level Metrics
- Project Level Metrics

4.1 Definition of Software Metrics

First, we must define "metric" and why we have used metrics as our major instrument in our dissertation. IEEE [16] defines a metrics as "A quantitative measure of the degree to which a system, component, or process possesses a given attribute". Software metrics are the means by which software engineers measure and predict aspects of those processes, resources, and products that are relevant to the software engineering activity. [19] Essentially, *software metrics* deals with the measurement of the software product and the process by which it is developed [28].

4.2 Importance of Software Metrics

Recently, the interests in software metrics have increased to a very high level in software industry. The project programmer and manager has begun to focus more on software productivity and software quality recently, due to increasing interests towards the new process improvement models, such as CMMI. This tendency resulted in looking for better technique of software development and software metrics during the process of development. In general terms, below are stated the main reasons that try to explain why software metrics become very important in software industry [42].

- Software metrics provide project managers more information on what is happening on the project development.
- Software metrics can help to better understand the development life cycle, especially, the design and architecture information of the software system.
- Software metrics can help to better understand the development process by applying the process evaluation during all stages of software development.
- Software design metrics can help to find out the errors in the software design at the early stage of software development life cycle, before causing further damage in terms of effort and time.
- Software metrics facilitate software testing activities.
- Software metrics can help to evaluate the software quality and provide an approximate cost estimate of the software project.
- Software metrics facilitate estimation and planning of new activities. By measuring current activity via metrics, it becomes easier to control the progress and improve the process to make it more cost-effective in the future.
- Software metrics can help to assess the effect of object oriented technology on the software development using solid quantitative evaluation criteria such as productivity, quality, lead time, maintainability, reusability.
- Software metrics can help to estimate the costs and benefits of different reuse strategies.
- Reusability metrics can help to assess the quality and reusability of software components and to detect potentially useful or reusable modules or components, saving valuable project resources.

4.3 Method-Level Metrics

In this section, we describe the method-level metrics we have selected for our case studies. At the beginning, we present a brief history of the metrics and then define them. The table below shows all the method-level metrics we have used in our experiments.

Table 2 : Method-Level Metrics

ABBREVIATION	***DESCRIPTION***
VG	McCabe Cyclomatic Complexity
MLOC	Method Lines of Code
NBD	Nested Block Depth
PAR	Number of Parameters

McCabe Cyclomatic Complexity (VG)

It is a measure of the complexity of a modules decision structure. It was developed by Thomas McCabe [37] and is used to measure the complexity of a program. It directly measures the number of linearly independent paths through a program's source code. The complexity analysis is also standalone and has been delivered as a product. It is a measure of reliability from the standpoint of what is required to test the system. It is a predictor of error.

Cyclomatic complexity is computed using the control flow graph of the program: the nodes of the graph correspond to the commands of a program, and a directed edge connects two nodes if the second command might be executed immediately after the first command.

It counts the number of flows through a piece of code. Each time a branch occurs (if, for, while, do, case, catch and the ?: ternary operator, as well as the && and || conditional logic operators in expressions), this metric is incremented by one. It is calculated for methods only. High values of this metric means that the application is very complex or at least that it have a large number of alternative flows.

Method Lines of Code (MLOC)

This metric defines the number of lines of code of all methods of a method, ignoring both blank and comment lines. Method lines of code are directly proportional to the program memory. Higher values of this metric lead to more memory footprint but also translate into more complex solution Detailed information on Lines of Code metrics is given in the following section under Lines Of Code Per Class (LOCC) metric.

Nested Block Depth (NBD)

The nested block depth metric measures the depth of conditional nesting in a method or module. The nesting depth is indicated by the width of the methods/modules flow graph. Therefore the metric is an indicator of complex control flow within the program. Deeply nested conditional statements increases the conceptual complexity of the code and are more likely to be error-prone. [5]

Number of Parameters (PAR)

This metric measures the number of parameters that are passed to a method. Objects with more than 4 parameters should be broken into separate algorithms for maintenance purposes.

4.4 Class-Level Metrics

In this section we describe the class-level metrics we have selected for our case studies. At the beginning, we present a brief history of the metrics and then define them. The table below shows all the class-level metrics we have used in our experiments.

Table 3 : Class-Level Metrics

ABBREVIATION	***DESCRIPTION***
DIT	Depth of Inheritance Tree
FOUT	Fan Out
LCOM	Lack of Cohesion of Methods
LOC_CLS	Lines Of Code for Class
NOF	Number of Attributes
NSC	Number of Children
NOTC	Number of Test Cases
NOM	Number of Methods
NORM	Number of Overridden Methods
NSF	Number of Static Attributes
NSM	Number of Static Methods
RFC	Response For Class
SIX	Specialization Index
TNOF	Total Number Of Fields
TNOM	Total Number Of Methods
WMC	Weighted methods per Class
PUB	Percentage of Public Data
CC	Cyclomatic Complexity
CBO	Coupling Between Objects

Depth Of Inheritance Tree (DIT)

This metric defines the depth of each class in the hierarchy within the object-oriented programming environment. In cases involving multiple inheritances, this metric will be the maximum length from the node to the root tree.

The deeper a class is in the hierarchy, the more methods it is likely to inherit, making it more complex. Deep trees as such indicate greater design complexity. As a positive factor, deep trees promote reuse because of method inheritance.

A high DIT has been observed to increase faults. However, it is not necessarily the classes deepest in the class hierarchy that have the most faults. The most fault-prone classes have been observed in a research to be the ones in the middle of the tree.[43] Root and deepest classes are consulted often, and due to familiarity, they have low fault-proneness compared to classes in the middle.

The nominal range for this metric is between 0 and 4. A compromise between the high performance power provided by inheritance and the complexity, which increases with the depth, must be found. A value of between 0 and 4 respects this compromise. A value greater than 4 would compromise encapsulation and increase complexity.

Fan Out (FOUT)

This metric is found by adding the number of other modules required and the number of data structures that are updated by the module being studied. The FOUT metric used in our experiments is an adaptation of Chidamber and Kemerer's [9] Coupling Between Object Classes (CBO) metric. It may be considered as a one-way version of CBO, as it does not include the classes it is used by.

A useful insight into the "object-orientedness" of the design can be gained from the system wide distribution of the class fan-out values. For example a system in which a single class has very high fan-out and all other classes have low or zero fan-outs, we really have a structured, not an object oriented system.

Lack of Cohesion of Methods (LCOM)

This definition of LCOM metric is slightly different by the original definition by Chidamber and Kemerer [9]. It was proposed by Henderson-Sellers. [15] The original LCOM metric is a count of the number of method pairs whose similarity is zero minus the count of method pairs whose similarity is not zero. Due to the facts that there are a large number of dissimilar examples with same LCOM value of zero, and that there is no guideline on the interpretation of any particular value, the new LCOM metric is normalized between zero and one.

The metric yields 0, in case all the fields of a class are accessed by all its methods. This condition indicates perfect cohesion. It yields 1 if each field of a class is accessed by exactly 1 method of this class. Conversely, this condition indicates complete lack of cohesion.

Cohesiveness of methods within a class is desirable, since it promotes encapsulation. Lack of cohesion implies classes should probably be split into two or more subclasses. Any measure of disparateness of methods helps identify flaws in the design of classes. Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.

Number of Children (NSC)

This metric defines the number of immediate subclasses subordinated to a class in the class hierarchy. It is a measure of how many subclasses are going to inherit the methods of the parent class.

High NSC indicates high reuse, since inheritance is a form of reuse. A large number of children (high NSC) may also mean improper abstraction of the parent class. If a class has too many children, it may indicate misuse of sub-classing. A class with many children may also require more testing.

NSC measures the breadth of a class hierarchy, where maximum DIT measures the depth. Depth is generally better than breadth, since it promotes reuse of methods through inheritance. High NSC has been found to indicate fewer faults. This may be due to high reuse, which is desired. Not all classes should have the same number of sub-classes. Classes higher up in the hierarchy should have more sub-classes then those lower down.

The nominal range for this metric is between 1 and 4. The upper and lower limits of 1 and 3 correspond to a desirable average. This will not stop certain particular classes being the kind of utility classes, which provide services to significantly more classes than 3.

Weighted Methods Per Class (WMC)

This metric defines the count of McCabe's cyclomatic complexity number [37] of all methods of a class. The number of methods and the complexity of methods involved is a predictor of how much time and effort is required to develop and maintain the class. The larger the number of methods in a class the greater the potential impact on children, since children will inherit all the methods defined in the class. Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse.

Lines Of Code Per Class (LOCC)

This metric defines the number of lines of code of all methods of a class, ignoring both blank and comment lines. A common basis of estimate on a software project is this metric. LOCC are used to create time and cost estimates. The LOCC estimate becomes the baseline to measure the degree of work performed on a project. Once a project is underway, the LOCC becomes a tracking tool to measure the degree of progress on a module or project. An experienced developer can gage a LOCC estimate based upon knowledge of past productivity on projects. The LOCC measurement becomes the barometer for the program's progress and productivity.

A standard definition and measurement technique for lines of source code is required to create a uniform basis of estimate for software projects. This measurement method must be independent of the operating system and applied uniformly to form a sound basis of estimate. Projects within a company will often use different methods for counting lines of code because a portable tool is not available for use on all operating systems.

Many programmers use a single brace or parenthesis on a line to block scope or code. This practice is very common, creates very readable code and is mandated by many commercial companies coding practices. A single character on a physical line may not create a line of code, which is representative of actual work performed by the programmer. This type of coding style will inflate LOCC metrics by 20 to 40 percent. That is why we ignore both blank and comment lines to obtain our line count.

Response For Class (RFC)

This metric is found by adding the number of methods (internal and external) of a class and the number of methods of other classes that are potentially available to this class.

If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding on the part of the tester. The larger the number of methods that can be invoked from a class, the greater the complexity of the class. A worst case value for possible responses will assist in appropriate allocation of testing time.

<u>Number of Attributes (NOF)</u>

This metric defines the total number of fields (attributes) in a class, including the class variables only, ignoring instance (internal – hidden) variables. It is used to count the average number of attributes for a class in the model. This information is useful in identifying the following potential problems:

- A class with too many attributes may indicate the presence of coincidental cohesion and require further decomposition, in order to better manage the complexity of the model.

- If there are no attributes, then serious attention must be paid to the semantics of the class, if indeed there are any. This may be a class utility rather than a class.

The nominal range for this metric is between 2 and 5. A high number of attributes (more than 10) probably indicates poor design, notably insufficient decomposition, especially if this is associated with an equally high number of methods. Classes without attributes are particular cases, which are not necessarily anomalies. These can be interface classes, for example, which must be checked.

<u>Number of Methods (NOM)</u>

This metric defines the number of methods in a class, including the external (class) methods only, ignoring the instance (internal – hidden) methods.

A class must have some, but not an excessive number of operations. This information is useful when identifying a lack of primitiveness in class operations (inhibiting re-use), and in classes which are little more than data types.

The nominal range for this metric is between 3 and 7. This range indicates that a class has operations, but not too many. A value greater than 7 may indicate the need for further object-oriented decomposition, or that the class does not have a coherent purpose. A value of 2 or less indicates that this is not truly a class, but merely a data construction.

<u>Number of Overridden Methods (NORM)</u>

This metric defines the total number of methods in the selected scope that are overridden from an ancestor class. The number of redefined operations plays a role in the specialization of the class and must be maintained in a proportion that continues to justify inheritance. Too many redefined operations imply too big a difference with the parent class and inheritance then makes less sense.

The nominal range for this metric is between 0 and 5. A class, which inherits services, must use them with a minimum of modifications. If this is not the case, the inheritance loses all meaning and becomes a source of confusion.

Number of Static Attributes (NSF)

This metrics defines the total number of static fields (attributes) in a class, including both the instance (internal – hidden) variables and class variables. Raising the number of Static Attributes translates into memory footprint increase and more complexity on the application.

Number of Static Methods (NSM)

This metric defines the total number of static methods in a class, including both the external (class) and instance (internal – hidden) methods. Static calls are faster than dynamic ones, translating into a performance increase. However, the abuse of static methods leads to a brittle solution that does not improve the reuse factor.

Specialization Index (SIX)

At the class-level, the number of classes inheriting a specific operation, the number of overridden methods (NORM) and new added methods can also be defined. Related to these measures, the Specialization Index (SIX) metric is defined as:

$$\boldsymbol{SIX} = \frac{\boldsymbol{NORM * DIT}}{\boldsymbol{NOM}}$$

where NOM represents the total number of methods for the class. This measure is useful in differentiating between implementation sub-classing (low values for SIX) and specialization sub-classing (high values of SIX).

Total Number Of Fields (TNOF)

This metric defines the total number of fields (attributes) in a class, including both the instance (internal – hidden) variables and class variables.

Total Number Of Methods (TNOM)

This metric defines the total number of methods in a class, including both the external (class) and instance (internal – hidden) methods.

<u>Percentage of Public Data (PUB)</u>

This metric defines the percentage of data that is public and protected data in a class. In general, lower values indicate greater encapsulation.

<u>Cyclomatic Complexity (CC)</u>

This metric is helpful to measure structural complexity. It is a measure of the complexity of a module's (component of a class, e.g. method) decision structure. It is the number of linearly independent paths. It counts the number of flows through a piece of code. Each time a branch occurs (if, for, while, do, case, catch and the ?: ternary operator, as well as the && and || conditional logic operators in expressions), this metric is incremented by one. It is calculated for methods only. High values of this metric means that the application is very complex or at least that it have a large number of alternative flows

<u>Coupling Between Objects (CBO)</u>

This metric defines the count of the number of other classes to which a class is coupled. Two classes are coupled when methods declared in one class use methods or instance variables defined by the other class. The uses relationship can go either way: both uses and used-by relationships are taken into account, but only once.

Multiple accesses to the same class are counted as one access. Only method calls and variable references are counted. Other types of reference, such as use of constants, calls to API declares, handling of events, use of user-defined types, and object instantiations are ignored. If a method call is polymorphic (either because of Overrides or Overloads), all the classes to which the call can go are included in the coupled count.

High CBO is undesirable. Excessive coupling between object classes is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse it in another application. In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design, and therefore maintenance is more difficult. A high coupling has been found to indicate fault-proneness. Rigorous testing is thus needed.

4.5 Package-Level Metrics

In this section we describe the package-level metrics we have selected for our case studies. The table below shows all the package-level metrics we have used in our experiments. Most of the metrics depend on the coupling metrics as defined by Robert Martin in "OO Design Quality Metrics, An Analysis of Dependencies" [25], and in his book named "Agile Software Development, Principles, Patterns and Practices". [24]

Table 4 : Package-Level Metrics

ABBREVIATION	***DESCRIPTION***
RMA	Richtmyer-Meshkov Abstractness
CA	Afferent Coupling
CE	Efferent Coupling
RMI	Richtmyer-Meshkov Instability
RMD	Normalized Distance from Main Sequence
NOC	Number of Classes
NOI	Number of Interfaces
LOC_PKG	Lines of Code per Package

Richtmyer-Meshkov Abstractness (RMA)

This metrics defines the number of abstract classes (and interfaces) divided by the total number of types in a package. The range for this metric is 0 to 1, with RMA = 0 indicating a completely concrete assembly and RMA = 1 indicating a completely abstract assembly.

According to how prone the package is to modification during the application's life cycle, it must be abstract to a greater or lesser extent. The more stable a package must be, the more abstract it must be, if it is to be extensible. Abstract packages that are extensible provide greater model flexibility.

This means that abstraction and instability must be jointly interpreted. This is synthesized by the *Abstraction/Instability* balancing metric, Normalized Distance from Main Sequence (RMD).

Afferent Coupling (CA)

This metrics defines the number of classes outside a package that depend on classes inside the package. It measures the number of types outside a package that depend on types within the package (incoming dependencies). High afferent coupling indicates that the concerned packages have many responsibilities.

Afferent and efferent coupling allows one to more effectively evaluate the cost of change and the likelihood of reuse. For instance, maintaining a module with many incoming dependencies is more costly and risky since there is greater risk of influencing other modules, requiring more thorough integration testing. Conversely, a module with many outgoing dependencies is more difficult to test and reuse since all dependent modules are required.

Concrete modules with high afferent coupling will be difficult to change because of the high number of incoming dependencies. Modules with many abstractions are typically more extensible, so long as the dependencies are on the abstract portion of a module.

Efferent Coupling (CE)

This metrics defines the number of classes inside a package that depend on classes outside the package. It measures the number of types inside a package that depends on types outside of the package (outgoing dependencies). High efferent coupling indicates that the concerned package is dependant.

Efferent coupling allows one to more effectively evaluate the cost of change and the likelihood of reuse. For instance, maintaining a module with many incoming dependencies is more costly and risky since there is greater risk of influencing other modules, requiring more thorough integration testing. Conversely, a module with many outgoing dependencies is more difficult to test and reuse since all dependent modules are required.

Richtmyer-Meshkov Instability (RMI)

This metrics defines the ratio of efferent coupling to sum of afferent (Ca) and efferent (Ce) coupling, i.e. the rate of instability of a package. A package is unstable if it depends more on other packages than they depend on it.

$$RMI = \frac{C_e}{(C_a + C_e)}$$

This metric does not have nominal values, since instability depends on what the package does. Certain packages must be unstable whilst others must not be unstable. This metric is an indicator of the package's resilience to change. The range for this metric is 0 to 1, with RMI = 0 indicating a completely stable package and RMI = 1 indicating a completely instable package.

A package is that much more unstable if it depends more on other packages than they depend on it. It is likely to change if these other packages change. Each value calculated for a given package must be compared to the values of the other packages. Not all packages have to be stable, since it must be possible for the application to evolve. If the user wishes the package to be stable, it must depend less on the other packages than they depend on it.

Normalized Distance from Main Sequence (RMD)

This metrics measures the balance between the abstraction and instability rates of the package, i.e. how far away a category is from this ideal. According to what function a package has to perform, it must be able to be unstable, in other words, often significantly or abstractly modified. It must be sufficiently general to be adaptable to widely diverse situations, either without being modified or with only minimal modifications. It is preferable to have a balance between these contradictory criteria.

For a package, the balance between abstraction and instability is obtained through the following expression:

$$\text{RMD} = | \text{Abstraction (A)} + \text{Instability (I)} - 1 |$$

An assembly squarely on the main sequence is optimally balanced with respect to its abstractness and stability. Ideal assemblies are either completely abstract and stable ($I = 0$, $A = 1$) or completely concrete and instable ($I = 1$, $A = 0$). The range for this metric is 0 to 1, with $D = 0$ indicating an assembly that is coincident with the main sequence and $RMD = 1$ indicating an assembly that is as far from the main sequence as possible. The picture in the report reveals if an assembly is in the zone of pain (I and A both close to 0) or in the zone of uselessness (I and A both close to 1).

Number of Classes (NOC)

This metrics defines the total number of classes inside a package. High values mean high memory footprint, higher complexity but high modularity too. Lower values can lead to poor application design but better system physical proprieties.

Number of Interfaces (NOI)

This metrics defines the total number of interfaces inside a package. Higher number of methods means more modularization (assuming two solutions with the same Method of Lines of Code) and this lead to a more readable solution but also mean more method calls. (that can greatly reduce performance)

Lines of Code per Package (LOC_PKG)

This metric defines the number of lines of code of all classes of a package, ignoring both blank and comment lines. This metric is obtained by summing the LOC_CLS metric values of all classes under the same package.

4.6 Project-Level Metrics

In this section we describe the two project-level metrics we have selected for our case studies. The table below shows all the project-level metrics we have used in our experiments. Most of the metrics depend on the coupling metrics as defined by Robert Martin in "OO Design Quality Metrics, An Analysis of Dependencies" [25], and in his book named "Agile Software Development, Principles, Patterns and Practices". [24]

Table 5 : Project-Level Metrics

ABBREVIATION	***DESCRIPTION***
NOP	Number of Packages
TLOC	Total Lines of Code

Number of Packages (NOP)

This metric defines the total number of packages in the project including both test and source packages, as for most of the cases, it is not possible to distinguish these two cases.

Total Lines of Code (TLOC)

This metric defines the number of lines of code of all classes of a project, ignoring both blank and comment lines. This metric is obtained by summing the LOC_PKG metric values of all packages in the project.

CHAPTER 5

A NEW MODEL ON TESTING EFFORT AND TESTABILITY

This chapter summarizes the model we have constructed as a result of our research in the first section, and also defines the construction steps of the model, in the second section. This second section of this chapter defines how we have constructed our model. It begins by including a brief information on the projects used and continues with the details about the experimental framework and statistical methodology. Statistical results and their assessments are presented afterwards. Regression analysis performed to compose the equations of our model is stated finally.

5.1 Our New Model

The mathematical equations of our model are given below that defines the expected amount of testing effort needed in terms of software metrics, and design parameters, belonging to source code which is based on design process directly.

5.1.1 Package-Level Model

The regression analysis we have performed at the package-level have produced the following equations for obtaining the expected metric values so that we can conclude that the packages are adequate to be tested properly and necessarily.

LOC_PKG_TEST =

$$\begin{aligned}&- 0.8528 * \mathbf{CA} + 95.0431 * \mathbf{NOI} - 1627.9391 * \mathbf{RMA}\\&+ 0.2802 * \mathbf{LOC\_PKG}\end{aligned}$$

NOTC_PKG_TEST =

$$\begin{aligned}&- 1.9674 * \mathbf{CE} + 11.9214 * \mathbf{NOI} - 207.6078 * \mathbf{RMA}\\&+ 32.5975 * \mathbf{RMD} + 23.1810 * \mathbf{RMI}\\&+ 0.0569 * \mathbf{LOC\_PKG}\end{aligned}$$

5.1.2 Class-Level Model

The regression analysis we have performed in class level have produced the following equations for obtaining the expected metric values and concluding that the source class-test class pair has a healthy relationship to be tested properly and necessarily.

LOC_CLASS_TEST =

$$\begin{aligned}&6.6672 * \mathbf{DIT} + 4.3128 * \mathbf{FOUT} + 5.0025 * \mathbf{NORM}\\&+ 2.2659 * \mathbf{NSF} - 0.9831 * \mathbf{RFC} - 16.7822 * \mathbf{SIX}\\&- 1.9227 * \mathbf{TNOF} + 0.7303 * \mathbf{WMC}\end{aligned}$$

LOC_CLASS_TEST_NEW =

$$\begin{aligned}&4.6033 * \mathbf{DIT} + 3.0515 * \mathbf{FOUT} + 9.4379 * \mathbf{LCOM}\\&+ 1.3550 * \mathbf{NSF} - 0.6470 * \mathbf{RFC} - 8.8597 * \mathbf{SIX}\\&- 1.0034 * \mathbf{TNOF} + 0.5780 * \mathbf{WMC}\end{aligned}$$

NOTC_CLASS_TEST =

$$\begin{aligned}&0.7822 * \mathbf{DIT} + 0.6295 * \mathbf{FOUT} + 1.6239 * \mathbf{NORM}\\&+ 0.8967 * \mathbf{NSF} - 0.1630 * \mathbf{RFC} - 4.6250 * \mathbf{SIX}\\&- 1.0067 * \mathbf{TNOF} + 0.4202 * \mathbf{TNOM} + 0.0967 * \mathbf{WMC}\end{aligned}$$

NOTC_CLASS_TEST_NEW =

$$3.2002 * \mathbf{DIT} + 2.1229 * \mathbf{FOUT} + 6.4686 * \mathbf{LCOM} + 0.9472 * \mathbf{NSF} - 0.4444 * \mathbf{RFC} - 6.2000 * \mathbf{SIX} - 0.7051 * \mathbf{TNOF} + 0.3992 * \mathbf{WMC}$$

5.2 Guidelines to Use Our Model

Considering the purposes of our study and consequences we have obtained evaluating and testing our model, as defined in CHAPTER 6, we may summarize that our model helps to:

- define our own understanding of testability,
- observe testability in terms of testing effort,
- identify probable non-conforming source class-test class pairs tested less than expected amount,
- identify probable non-conforming source packages tested less than expected amount,
- identify the major source based metrics affecting the testing effort,
- identify the major design parameters affecting the testing effort,
- define guidelines to alter testability level,
- perform and evaluate software design according to testability and testing effort.

Our model defines mathematical equations for obtaining the expected values of the test metrics, in two different levels, i.e. class and package levels. One may make use of our model and equations in two different points of view: either during design phase, i.e. or after the implementation phase.

Considering the use of the model in the design phase of the project, to be able to alter the expected values, one has to increase the corresponding metric value with a negative sign and decrease the metric value with a positive sign. Altering the expected values also leads to testability guidelines for software project staff, as decreasing the expected value means you have spend less amount of effort to test the corresponding class or package, hence increase its testability. For example, to be able to test a package with less testing effort, i.e. smaller expected value of testing metrics, you need to decrease the size of your package, which is obvious, increase the abstractness of you package, decrease the number of interfaces and increase the level of afferent coupling in your package. The examination of the equations indicates strong correlation with the explanations given in the corresponding metric definitions, on to have a high or a low value o the metrics to have a more stable structure of software design.

Making use of our model and equations after the implementation phase is completed means that one aims to identify probable non-conforming source-test pairs, which indicate that the testing effort is not adequate for these pairs. In order to be able to make use of our model and equations after the implementation phase is completed, one has to follow the steps described below:

- Identify the test classes and packages, and the corresponding source-test class pairs for class-level analysis.
- Use metrics plug-in (version 1.3.6 or higher) by Frank Sauer under Eclipse platform or any other metrics plug-in that measure the metrics necessary for the equations and calculation.
- Calculate the expected class and package level test metric values using the equations. The proposed two new metrics in class level aim to decrease the number of testing parameters, i.e. LOC and NOTC into one parameter, either LOC_CLASS_TEST_NEW or NOTC_CLASS_TEST_NEW. These two metrics have a strong correlation. Therefore, one may choose to use either one of the new metrics, or the two older metrics (LOC_CLASS_TEST and NOTC_CLASS_TEST) to be able to assess the testing effort in the project.
- Compare the observed values in the project and the expected values given by the equations of our model, for both class and package levels. Calculate the divergence ratios of each source-test class pairs (source classes having corresponding testing class) and source-test package pairs (source packages having corresponding testing packages), which define the ratio how much the expected and the observed values differ.
- Define your own maximum allowed level of divergence ratio. This maximum level aims to identify the class and package pairs that differ more than the allowed divergence ratio level. This value was chosen as “0.5” in our validation process. It may vary according to the context of the project, i.e. characteristics of your organization and the type, owner and final user of the project.
- Pay attention to the classes and packages that have divergence ratios higher than the threshold value you have determined. Try to decrease the divergence ratio by decreasing the difference between expected and observed metric values. Examining the equations, try to increase the values

of the source metric values that have positive coefficients and decrease the values of the source metric values that have negative coefficients.

5.3 Our Approach

The major tools we will use to accomplish our goals can be summarized as follows:

- Source-based software metrics
- Open-source Java projects
- JUnit testing framework
- Multiple linear regression
- Eclipse platform and metric plug-in [12]
- Oracle 11g RDBMS and statistics package [48]
- Spearman's [40] rank-order correlation coefficient

Our study primarily addresses software projects developed within the OO environments using Java programming language, and tested with the JUnit framework. Nevertheless, it may be applicable and useful in related environments, as well. This will be discussed in detail in CHAPTER 7.

We have used source-based metrics to assess their effects on the effort that is required for testing process. A software metric plugin is the primary tool of our study, which helps to measure the software product and the process by which it is developed. A source-based metric is a metric calculated using the source code of the software product. All metrics bear and represent an important design factor of the software project. In this way, we will be able to observe the relationships among certain design factors and our primary concern factors, testability and testing effort.

Our dissertation uses a large set of metrics, which are commonly used to assess object-oriented software systems written in a common language, namely Java. The software language chosen is Java as it is widely used in web-based software projects. In addition, the number of open-source Java projects available on the web is substantially large. This provides the main measurement data for our dissertation.

The reason why we have focused on Object-Oriented environment is simple. Object-Oriented programming is a popular and commonly used programming paradigm, which has not been examined with respect to older paradigms. In addition, OO software systems are widely used in web-based systems, which provide easy data measurement facility, as well.

Using source-based metrics to compose our assessment has some advantages over other methods. It is practical to use source-code rather than design documents, as different projects may have different types of documents. Practice shows that, documentation is always in the second place with respect to accomplishing the project, which results in out-of-date design documents, or even lacking or no documents, at all. However, source code is a direct mirror of the implementation, which also contains aspects of other stages of software development. Automatic processing of source code, thus, is much simpler with respect to other methods.

All software projects may not have recorded time, effort and cost data. Even if two different projects may have the same level of measurement data, this does not clearly indicate that the two projects can be categorized into the same class. The process of the software company, human resource expertise levels, and many other factors may obviously affect the results. Thus, it is better to use source-based metrics, as our assessment will be independent of the other factors.

We have used open-source projects written in Java language belonging to two popular open-source frameworks, Apache and JBoss. The projects of these two organizations that are subject to our research are unit tested at the class level using the JUnit testing framework [47]. This framework helps to create classes that are capable of unit testing a part of the system. An ideal situation would be to have a test class for every class of the system. As this is impossible and unnecessary in practice, we have tried to obtain the projects that have the biggest number of source class-test class pairs.

We have used JUnit documentation [47] to determine the mechanism to detect the corresponding test class for every system class. The JUnit documentation suggests that test classes should be named after the class they test, by appending “Test" to the name of the class. Although this convention is generally used in both our study systems, we had to consider other conventions, as well, to associate a class and its test class in an automated way. Appending “Test" as a prefix before the name of the class, appending “Test”, “TestCase" and “TestSuite” as a suffix after the name of the class were the other two commonly used convention we have noticed and taken into account.

The Eclipse tool platform [31] has been used to calculate the source-based class and project level metrics. An existing plug-in for Eclipse, the “Eclipse metrics plug-in (version 1.3.6)" by Frank Sauer [12], was extended by Magiel Bruntink [7] to calculate the set of metrics we are interested, including dLOC_CLS and dNOTC test metrics. Functionality to calculate many metrics was already present in the original version of the plug-in. Magiel added support for the FOUT, RFC and dNOTC metrics and adapted the existing implementations of the field and method counts to better reflect the existing NOF and NOM metrics of Magiel's version. The Eclipse platform extension mechanism allowed quick integration of the new metrics into the plug-in.

Both source classes and test classes are measured using the Eclipse plug-in, which stores the resulting values in an XML file, and then in a relational database via a small Java program. The original plug-in offered exporting of the results to an XML file. The size of our case studies resulted in XML files that were very large and hard to process. Therefore, we have moved the data in these XML files into an Oracle database [48]. The use of a relational database made it possible to efficiently store, access and statistically experiment the data. Using an XML processing class, we have stored the results in the XML files directly to the Oracle database [48].

The major statistical function we have used to examine our metrics data is Spearman's [40] rank-order correlation coefficient, ρ. Spearman's rank correlation coefficient is a non-parametric measure of correlation – that is, it assesses how well an arbitrary monotonic function could describe the relationship between two variables, without making any assumptions about the frequency distribution of the variables. [40]

We have used PL/SQL and embedded Extended Statistical Function Set of Oracle 11g RDBMS [48] to calculate the correlations among the metric values and extract the required information from the raw data we have loaded from the XML files. Multiple linear regression was the key mathematical tool to obtain relationships among the software metrics and related test metrics. As many other complicated forms such as polynomial, logarithmic, exponential equations are applicable, we have limited our regression on the simplest form due to calculation simplicity and availability.

To be able to assess whether or not the testing effort and cost consumed is adequate is a critical matter this dissertation aimed to answer by composing new way to evaluate the links between software design parameters and testing effort via source-based metrics. Software projects belonging to two different open-source frameworks helped us to achieve our goals.

In our dissertation, we have presented significant associations, relationships and properties of source based metrics in many different levels, i.e. method, class, package and project. We have proposed new test metrics in various levels. We have found significant associations between the source-based metrics and the test suite metrics. We have also examined the relationships among the source-based metrics, as well to observe how different metrics belonging to different design parameters affect each other.

We have also performed regression analysis in both class and package levels, and proposed new equations for obtaining the expected metric values so that we can conclude that the packages are adequate, and source class-test class pair has an adequate relationship to be tested properly and necessarily.

We have composed a new model of testing effort and testability via the proposed equations using the available object-oriented software metrics. The new model we have proposed is significant, as there are only a few models in the literature proposed on testing effort and testability concept. We have tested our model on new open-source projects, which have not been used in the model construction part of our study. The results of testing our model validated the strength and success of our model to define expected values for the test metrics, which help us to identify probable non-conforming testing components (packages or test class pairs) in our project. In addition, we have interpreted the equations to utilize the model in the phrasing of guidelines for testability.

5.4 Construction Of The Model

This section defines how we have constructed our model. It begins by including a brief information on the projects used and continues with the details about the experimental framework and statistical methodology. Statistical results and their assessments are presented afterwards. Regression analysis performed to compose the equations of our model is stated finally.

Testability is a very important concept, but it is also very difficult and subjective to define in a mathematical relationship, as it is not possible to link testability and testing effort with other software concepts directly. Before using open-source Java projects as the main repository for our study, we tried to extract information using another repository belonging to NASA.

The main source for our previous data was the Project Repository of NASA Independent Verification and Validation (IV & V) Facility Metrics Data Program [30]. Two different Object Oriented Software Projects of different size and programming languages seemed adequate for our research, as they were the ones developed in the object-oriented environment and had significant scale in size to be evaluated.

We applied a similar methodology to obtain a model, but we had to give up working on the repository due to some reasons. The model aimed to measure the testability indices of software components, i.e. methods and classes, for the assessment of testability using design parameters in object-oriented design. The proposed model in the NASA study needed to be validated using structural and functional information from different software projects, as it was based on two mid-sized object-oriented software projects developed within the same organization and contains a limited set of object-oriented software metrics.

The model in the NASA study had a view of testability from a certain assumption that a software project's testability may be obtained by normalizing the error density (count of defects per lines of code) it faces during live usage after delivery, i.e. the number of defects found in the delivered software system,. The results did not indicate that this was a wrong assumption from statistical perspective, but we had to have more data to get better correlation and significance results.

The major problem with NASA study was the fact that the projects under analysis are few in number and small in size. In addition, the error data available did not give detailed information about the defects in the software, so that our hypothesis that transformed the error rate in time into testability index might not be correct under all circumstances.

As we could not obtain more projects and significant data, we decided to stop working on NASA data and look for other possible sources. Open source Java projects were found as the perfect candidate, lacking error log data different from NASA projects. Therefore, instead of trying to compose a mathematical definition of testability, we have decided to map the relationships between testing effort and design parameters via source metrics, and then, if possible, relate them to testability. The two primary sources for our study were Apache Software Foundation [44] and JBoss Software Community [45], both of which contain and host development of many open-source projects. Both software communities are suitable candidates for study for a number of reasons. First, most of the projects that are currently being developed or have been developed were coded in Java language, the most popular object-oriented programming environment. Nearly all projects hosted are open-source, so their source codes and documentation is public to anyone.

Most of the major projects, the ones chosen for our study, are unit tested at the class level, which is the perspective we assume in this dissertation. Both communities use the JUnit testing framework [47] to implement their test suites. Both communities have their own quality standards such as coding standard and design guidelines, which means all projects have a quality baseline, i.e. documentation, and share the same guidelines. Finally most of the projects taken into our study are middle-scale projects due to their lines of code count and number of classes and packages they contain.

5.4.1 Apache Projects

Formerly known as the Apache Group, the Apache Foundation [44] has been incorporated as a membership-based, not-for-profit corporation in order to ensure that the Apache projects continue to exist beyond the participation of individual volunteers.

The Apache Software Foundation provides organizational, legal, and financial support for a broad range of open source software projects. The Foundation provides an established framework for intellectual property and financial contributions that simultaneously limits contributor's potential legal exposure.

Through a collaborative and meritocratic development process, Apache projects deliver enterprise-grade, freely available software products that attract large communities of users. The pragmatic Apache License makes it easy for all users, commercial and individual, to deploy Apache products.

Table 6 : Details of Projects Used

Project Name	**Total Line Count (KLOC)**	**Total Test Line Count (KLOC)**	**Number of Classes**	**Number of Packages**
Apache Ant	116	13.58	1362	78
Apache Lucene	109.3	23.36	1119	84
Apache Geronimo	163.6	12.27	2064	287
Apache Mina	21.55	3.16	330	40
Apache Wicket	120	7.96	2060	242
Apache JackRabbit	124.3	20.61	1241	107
Apache ActiveMQ	118.1	17.07	1473	114
Apache Maven	34.6	3.63	287	70
Apache ODE	53.5	2.70	896	94
Apache OJB	192.1	26.65	1424	87
Apache OpenEJB	122.7	3.01	1712	91
Apache Struts	43.9	10.44	741	68
Apache Tapestry	62.9	14.21	967	72
JBoss Cache	102.4	35.44	882	63
JBoss Drools	156.7	31.23	1558	112
JBoss Richfaces	112.8	9.71	1315	101
JBoss ESB	66.3	10.07	820	148

Since all projects are subprojects of the Apache Software Foundation, their development process is a derivative of the Apache project. In turn, the Apache project is a derivative of the popular open source model. Typically, an open source project consists of a number of contributors from around the world, who communicate and work together via the Internet. The open source model is not a full-edged development methodology. Its main concerns are project management and adherence to a number of beliefs, including the free availability of source code. As such, most of the projects follow the same coding guideline presented by Sun, "Code Conventions for the Java ^{TM} Programming Language.[35]

For all Apache projects, the programmers develop JUnit [47] test cases during development, and run these tests nightly. Additionally, the functional correctness of the entire system is verified every night by running Ant scripts in a typical production environment. There is no explicit testing criterion; test cases are created based on the preference of the programmers. Consequently, no measurement of the level of compliance to the testing criterion is done. Bug reports are again used as a source of test cases. In addition, for most of the projects, the source code and documentation is kept in a public CVS repository, which can be read by anyone. [7]

Below are given details on the Apache [44] and JBoss [45] projects, together with their project version number and date of distribution, used in the experiments. All projects have detailed information (documentation, source and binary codes downloadable) on their own web sites accessible from the owner organizations main web sites.

5.4.1.1 Ant (v 1.7.0 - 19.12.2006)

Apache Ant is a Java-based build tool. A build tool is used to automate many tasks related to the source code of a program, like compilation, execution and packaging. Many other tools exist that solve the same problem, including well-known UNIX tools like Make. Ant aims at being portable, i.e. capable of running on multiple platforms, and at being easily extensible by Java.

5.4.1.2 Geronimo (v 2.0.2 - 19.10.2006)

The goal of the Geronimo project is to produce a server runtime framework that pulls together the best Open Source alternatives to create runtimes that meet the needs of developers and system administrators. The most popular distribution is a fully certified Java EE 5 application server runtime.

5.4.1.3 Lucene (v 2.2.0 - 19.06.2007)

Apache Lucene is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform.

5.4.1.4 Mina (v 1.1.4 – 29.10.2007)

Apache MINA is a network application framework, which helps users to develop high performance and high scalability network applications easily. It provides an abstract event-driven asynchronous API over various transports such as TCP/IP and UDP/IP via Java NIO. MINA is a simple yet full-featured network application framework providing many useful and new properties, such as unified API, JMX manageability, etc.

5.4.1.5 Wicket (v 1.3.0-rc1 - 09.11.2007)

Wicket is one of the most recent in a long line of Java web development frameworks and stands on the shoulders of many that have come before it. Wicket is a component-based framework, which puts it in stark contrast to some of the earlier solutions to the sometimes-monotonous task of web programming.

Like other frameworks, Wicket builds on top of Sun's servlet API; however, unlike frameworks like Struts or Spring MVC, the developer using Wicket is not responsible for the request/response nature that is inherent with the web and Servlets. Instead of building controllers that must service many users and threads simultaneously, taking in requests, returning responses, and never storing any state, the Wicket developer thinks in terms of stateful components. Instead of creating a controller or action class, he or she creates a page, places components on it, and defines how each component reacts to user input.

5.4.1.6 JackRabbit (v 1.3.3 - 04.10.2007)

Apache Jackrabbit is a fully conforming implementation of the Content Repository for Java Technology API (JCR). JCR is the acronym of the JSR 170: Content Repository for Java technology API, a standard interface for accessing content repositories. A content repository is a hierarchical content store with support for structured and unstructured content, full text search, versioning, transactions, observation, and more. Typical applications that use content repositories include content management, document management, and records management systems.

5.4.1.7 ActiveMQ (v 4.1.1 - 23.03.2007)

Apache ActiveMQ is one of the most popular and powerful open source message broker and enterprise integration patterns providers. Apache ActiveMQ is a fast solution, that supports many Cross Language Clients and Protocols, comes with easy to use Enterprise Integration Patterns and many advanced features while fully supporting JMS 1.1 and J2EE 1.4. Apache ActiveMQ is released under the Apache 2.0 License.

5.4.1.8 ODE (v 1.1 - 27.08.2007)

Apache ODE (Orchestration Director Engine) executes business processes represented in the WS-BPEL standard. It talks to web services for sending and receiving messages, handling data manipulation and error recovery as described in the process definition. It supports both long and short living process executions to orchestrate all the services that are part of the application.

5.4.1.9 OpenEJB (v 3.0-beta1 - 28.08.2007)

Apache OpenEJB is an embeddable and lightweight EJB 3.0 implementation that can be used as a standalone server or embedded into Tomcat, JUnit, TestNG, Eclipse, IntelliJ, Maven, Ant, and any IDE or application. OpenEJB is included in Apache Geronimo, IBM WebSphere Application Server CE, and Apple's WebObjects.

5.4.1.10 Struts (v 2.0.11 - 21.09.2008)

Apache Struts 2 is an elegant, extensible framework for creating enterprise-ready Java web applications. The framework is designed to streamline the full development cycle, from building, to deploying, to maintaining applications over time. Apache Struts 2 was originally known as WebWork 2. After working independently for several years, the WebWork and Struts communities joined forces to create Struts^2. This new version of Struts is simpler to use and closer to how Struts was always meant to be.

5.4.1.11 Tapestry (v 5.0.6 - 17.10.2007)

Tapestry is an open-source framework for creating dynamic, robust, highly scalable web applications in Java. Tapestry complements and builds upon the standard Java Servlet API, and so it works in any servlet container or application server. Tapestry divides a web application into a set of pages, each constructed from components. This provides a consistent structure, allowing the Tapestry framework to assume responsibility for key concerns such as URL construction and dispatch, persistent state storage on the client or on the server, user input validation, localization/internationalization, and exception reporting. Developing Tapestry applications involves creating HTML templates using plain HTML, and combining the templates with small amounts of Java code using (optional) XML descriptor files.

5.4.1.12 Maven (v 2.0.7 - 17.06.2007)

Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information. Maven allows a project to build using its project object model (POM) and a set of plugins that are shared by all projects using Maven, providing a uniform build system. Once the user familiarizes with how one Maven project builds, she automatically knows how all Maven projects are built saving one immense amounts of time when trying to navigate many projects.

5.4.1.13 ObJect Relational Bridge (OJB) (v 1.0.4 - 31.12.2005)

Apache ObJectRelationalBridge (OJB) is an Object/Relational mapping tool that allows transparent persistence for Java Objects against relational databases. OJB uses an XML based Object/Relational mapping. The mapping resides in a dynamic MetaData layer, which can be manipulated at runtime through a simple Meta-Object-Protocol (MOP) to change the behaviour of the persistence kernel.

5.4.2 JBoss Projects

JBoss.org [45] is owned by Red Hat [46], a popular provider of Linux and open source technology. The JBoss community consists of individuals and companies from all over the world who participate as users, testers, developers, writers and speakers for the projects. The unifying goal and vision is to develop the best possible Java Enterprise Middleware in open source, available for anyone to use with no license fees. As part of the community, one will have plenty of opportunities to learn from other experienced developers and users who share their desire for success.

JBoss community projects sit between the application code and the operating system to provide services such as persistence, transactions, messaging and clustering. Implementing this software in Java allows it to run on many different operating systems, giving the end-user the flexibility to develop and deploy applications. The aim is to regularly release stable versions together with documentation for use in cutting-edge application development.

JBoss projects are developed in open source in order to benefit from the high level of innovation and extensive testing provided by online communities. JBoss has chosen the business-friendly LGPL as the main license to ensure that one can safely use them to develop and deploy applications whilst keeping the source code private. The user may even keep changes made to the JBoss project source code private as long as he does not distribute the resulting binaries.

5.4.2.1 Cache (v 1.3.0-rc1 - 19.08.2007)

JBoss Cache is a tree-structured, clustered, transactional cache. It is the backbone for many fundamental JBoss Application Server clustering services, including - in certain versions - clustering JNDI, HTTP and EJB sessions. JBoss Cache can also be used as a standalone transactional and clustered caching library or even an object oriented data store. It can even be embedded in other enterprise Java frameworks and application servers such as BEA WebLogic or IBM WebSphere, Tomcat, Spring, Hibernate, and many others.

5.4.2.2 Drools (v 4.0.3 – 22.10.2007)

Drools (JBoss Rules) is a business rule management system (BRMS) and an enhanced Rules Engine implementation, ReteOO, based on Charles Forgy's Rete algorithm tailored for the Java language. Drools is an open source and standards-based business rules engine for easy business policy access, change, and management. It is a fast, highly efficient rules engine that makes it easy for a business analyst or auditor to view business rules, as they are encoded in IT application infrastructures, to verify that the encoded rules indeed implement the documented business policies. It also supports a variety of language and decision table inputs, making it easy to quickly modify business policies to respond to opportunities and competitive threats.

5.4.2.3 Richfaces (v 3.1.2.GA – 17.10.2007)

RichFaces is a rich component library for JSF and an advanced framework for easily integrating AJAX capabilities into business application development. The RichFaces components come ready to use out-of-the-box, so developers can immediately save time in taking advantage of component features to create Web applications that provide greatly improved user experience more reliably and more quickly. RichFaces also includes strong support for the skinnability of JSF applications. RichFaces also takes full advantage of the benefits of the JSF framework including lifecycle, validation, and conversion facilities, along with the management of static and dynamic resources.

5.4.2.4 ESB (v 4.2.1.GA - 12.10.2007)

ESB is a new Enterprise Application Integration (EAI) tool. It contains the following EAI stacks: Business Process Monitoring, Integrated Development Environment, Human Workflow User Interface, Business Process Management, Connectors, Transaction Manager, Security, Application Container, Messaging Service, Metadata Repository, Naming and Directory Service, Distributed Computing Architecture.

An ESB is part of an SOA infrastructure. However, SOA is not simply a technology or a product: it's a style of design, with many aspects (such as architectural, methodological and organizational) unrelated to the actual technology. Nevertheless, obviously at some point, it becomes necessary to map the abstract SOA to a concrete implementation and that's where JBoss ESB comes in to play.

5.4.3 Proposal of New Test Metrics

We have proposed new metrics in addition to the LOC and NOTC metrics, in class and package levels, in accordance with the objectives of our study. The LOC_CLS and NOTC_CLS metrics and the dLOC_CLS and dNOTC_CLS metrics have been previously defined for class level only. We have defined new versions of these metrics in package and project levels. Method-level versions are not possible, as most of the test cases do not test the corresponding source methods, they aim to test the functionality of the source class as a whole, not method by method.

In class level, we have defined three new metrics, one for the source classes, two for the test classes. In package level, we have defined only two new metrics, as test packages and source packages are not separate for most of the projects.

We have decided to obtain new metrics, assuming that a multiple linear equation exists among the current metrics. Then, a multiple linear regression has been used to get the coefficients of this equation. The multiple linear regressions establish a relationship between dependent variables and multiple independent variables. The regression equation takes the form:

$$y = \beta_0 + \beta_1 x_1 + \ldots\ldots + \beta_m x_m$$

where “x”s represent the independent variables, i.e. our current metrics, “y” is the dependent variable, i.e. our new metric to be proposed and “β”s represent the regression coefficients which indicate the net effect the independent variable on the dependent variable, holding the remaining variables in the equation constant. Component-wise effect may be speculated and respective component weightings (CW) may be fixed using the regression equation. Thereby, the CWs of individual design parameters have been calculated in terms of the regression coefficient β.

Using the coefficients given in Table 8 that summarize the statistical results of our study, we propose the class and package metrics shown in Table 7.

Table 7 : Proposed Metrics

Class-Level Metrics	Package-Level Metrics	Project-Level Metrics
LOC_CLS_NEW	LOC_PKG	
dLOC_CLS_NEW	dLOC_PKG	dLOC_PRJ
dNOTC_CLS_NEW	dNOTC_PKG	dNOTC_PRJ

Table 8 : Weightage Coefficients of Proposed Metrics

	NEW LOC METRIC	NEW NOTC METRIC
DIT	0.06	0.08
FOUT	0.34	0.24
LCOM	0.26	0.08
LOCS	0.42	0.29
NOF	0.31	0.09
NOM	0.36	0.18
NORM	0.10	0.10
NSC	0.08	0.08
NSF	0.15	0.11
NSM	0.04	0.16
RFC	0.37	0.29
SIX	0.06	0.08
TNOF	0.33	0.15
TNOM	0.37	0.24
WMC	0.41	0.29

New Class and Test metrics proposed use the coefficients given in the table above as weightings. These table values have been obtained from Table 8, which shows correlation analysis results of Class Level Metrics for the case considering all projects as one single project. New source class metrics use the metric values of the source classes, and new test class metrics use the metric values of the test classes.

For example, the value of the LOC_CLS_NEW metric is calculated as:

LOC_CLS_NEW = 0.06 * **DIT** + 0.34 * **FOUT** + 0.26 * **LCOM** + 0.42 * **LOCS** + 0.31 * NOF + 0.36 * **NOM** + 0.10 * **NORM** + 0.08 * NSC + 0.15 * **NSF** + 0.04 * **NSM** + 0.37 * **RFC** + 0.06 * **SIX** + 0.37 * **TNOM** + 0.4 * **WMC**

5.4.4 Experimental Framework

Empirical study within the field of software engineering is relatively rare. The study of Magiel Bruntink [7] aims to answer the same question like our dissertation. During our studies, Magiel shared his adaptation of the metrics tool and helped us compose our experimental framework. We will follow his framework guideline, but will extend and modify the mathematical calculations with more functions and different points of view.

Goal Question Metric / MEtric DEfinition Approach (GQM/MEDEA) framework proposed by Basili seems appropriate for our study. [6] First, we define the goal of our experiments:

Goal: To assess the capability of the proposed source-based metrics to predict the testing effort.

Next, we describe our perspective on the goal, and relevant factors of the environment, the context of the experiments.

Perspective: We evaluate the source-based metrics at the class, package and project levels, and limit the testing effort to the unit testing of classes. Thus, we try to figure out and assess whether or not the values of the source-based metrics can predict the required amount of effort needed for unit testing a class.

Environment: The experiments are targeted at Java systems, which are unit tested at the class level using the JUnit testing framework [47].

To help us translate the goal into measurements, we pose questions that pertain to our goal:

Question 1: Are the values of the source-based metrics for a class associated with the size of the corresponding test suite, i.e. the required testing effort for that class?

We use the two metrics proposed by Bruntink [7], which are dLOCC (Lines Of Code for Class) and dNOTC (Number of Test Cases) metrics to indicate the size of a test suite. The "d" prepended to the names of these metrics denotes that they are the dependent variables of our experiment.

The dLOC metric is defined similar to the LOC metric. The dLOC metric is applicable because typical use of JUnit [47] would be to test a class using a single test class. The dNOTC metric is calculated by counting the number of invocations of JUnit `assert' methods [47] that occur in the code of a test class. JUnit [47] provides the tester with a number of different `assert' methods, for example `assertTrue', `assertFalse' or `assertEqual'. The operation of these methods is the same: the parameters passed to the method are tested for compliance with some condition, depending on the specific variant. For example, "assertFalse" tests whether or not its parameter evaluates to "false". If the parameters do not satisfy the condition, the framework generates an exception that indicates a test has failed. Thus, the tester uses the set of JUnit "assert" methods to compare the expected behavior of the class-under-test to its current behavior. As a result, by counting the number of invocations of "assert" methods, we count the number of comparisons of expected and current behavior. We consider the latter to be an appropriate definition of a test case. [7]

The next question derives the hypotheses that our experiments will test. The question is:

Question 2: Are the values of the source-based metrics for a class associated with the dLOCC and dNOTC metrics of the corresponding test suite?

Hypotheses:

$\mathbf{H_0}$(x; y): There is no association between design metric x and test suite metric y,

$\mathbf{H_1}$(x; y): There is an association between design metric x and test suite metric y,

where x ranges over our set of source-based metrics, and y is either the dLOC or dNOTC of the associated test suit.

The systems that are subject to our experiments (Apache and JBoss) both are unit tested at the class level using the JUnit testing framework [47]. This framework helps to create classes that are capable of unit testing a part of the system. An ideal situation would be to have a test class for every class of the system. As this is impossible and unnecessary in the practice, we have tried to obtain the projects that have the most testing-tested class pairs.

The Eclipse tool platform [31] was used to calculate the source-based class and project level metrics. An existing plug-in for Eclipse, the “Eclipse metrics plug-in (version 1.3.6)" by Frank Sauer [12], was extended by Magiel Bruntink [7] to calculate our set of metrics for a given system, including dLOC_CLS and dNOTC metrics. Functionality to calculate many of our metrics was already present in the original version of the plug-in. Magiel added support for the FOUT, RFC and dNOTC metrics and adapted the existing implementations of the field and method counts to better reflect the existing NOF and NOM metrics of Magiel’s version. The Eclipse platform extension mechanism allowed quick integration of the new metrics into the plug-in.

We used the plug-in to measure the test classes, and their two corresponding metrics, i.e. their dLOC_CLS and dNOTC values. Both source classes and test classes are measured using the Eclipse plug-in, which stores the resulting values in an XML file, and then in a relational database via a small Java program. The original plug-in offered exporting of the results to an XML file. The size of our projects resulted in XML files that were very large and hard to process. Nevertheless, we have moved the data in these XML files into an Oracle database [48]. The use of a relational database made it possible to efficiently store, access and statistically experiment the data. Using an XML processing class, we have stored the results in the XML files directly to the Oracle database [48].

Finally, the original plug-in operates in an interactive mode, i.e. the Eclipse platform user-interface. As the visual results were not our primary objective, we have used the “head-less” operation mechanism of Eclipse platform using Ant scripts which is invoked from the command line, to be able to run and extract the metric calculations without the user-interface attached, effectively handing control to the plug-in itself.

The calculation process itself is straightforward. Assuming the plug-in has been invoked from the command line, i.e. it is operating in batch mode, the following steps occur:

- The hierarchy of Java elements, i.e. methods, types, classes and packages, is traversed.
- For each Java element:
    - The appropriate metrics are calculated for the category of the Java element.
    - The metric values are stored in a data structure in memory.
    - The stored metric values are exported to the XML file.

The calculation of our metrics set in the plug-in uses the Eclipse Java parser to obtain an abstract syntax tree (AST) representation of the Java element. Subsequently, the AST is used to calculate the actual metric value. Many of our metrics traverse the AST using visitors, which originate from the visitor design pattern, defined by [13]. The support of the Eclipse platform for this kind of traversal allowed us to implement the new metrics with little effort. [7]

5.4.5 Statistical Methodology

The major statistical function we use to examine our metrics data is Spearman's [40] rank-order correlation coefficient, ρ, which we calculate for each source-based metric of the system classes and both the dLOC and dNOTC metrics of the corresponding test classes. In statistics, Spearman's rank correlation coefficient, named after Charles Spearman [40] and often denoted by the Greek letter ρ (rho) or as $r_{s,}$ is a non-parametric measure of correlation – that is, it assesses how well an arbitrary monotonic function could describe the relationship between two variables, without making any assumptions about the frequency distribution of the variables. [40]

Siegel and Castellan defines it as a measure of association between two variables that are measured in at least an ordinal scale [32]. The measurements are ranked according to both variables. Subsequently, the measure of association is derived from the level of agreement of the two rankings on the rank of each measurement. The value of ρ can range from -1, indicating perfect negative correlation, to 1, indicating perfect positive correlation. A ρ value of 0 indicates no correlation. [7]

The ρ statistic allows its application even if the distribution of the data is not known. This fact is the main motivation for our use of ρ, since we indeed lack knowledge about the distribution of the metric values.

The modern approach to testing whether an observed value of ρ is significantly different from zero (we will always have $1 \geq \rho \geq -1$) is to calculate the probability that it would be greater than or equal to the observed ρ, given the null hypothesis, by using a permutation test. This approach is almost always superior to traditional methods [40], unless the data set is so large that computing power is not sufficient to generate permutations, or unless an algorithm for creating permutations that are logical under the null hypothesis is difficult to devise for the particular case (but usually these algorithms are straightforward).

Although the permutation test is often trivial to perform for anyone with computing resources and programming experience, traditional methods for determining significance are still widely used. The most basic approach is to compare the observed ρ with published tables for various levels of significance. This is a simple solution if the significance only needs to be known within a certain range or less than a certain value, as long as tables are available that specify the desired ranges. However, generating these tables is computationally intensive and complicated mathematical tricks have been used over the years to generate tables for larger and larger sample sizes, so it is not practical for most people to extend existing tables. [40]

Before proceeding to calculate ρ values, we need to find the corresponding test class for every system class. The JUnit documentation [47] suggests that test classes should be named after the class they test, by appending “Test" to the name of the class. Although this convention is generally used in both our study systems, we had to consider other conventions, as well, to associate a class and its test class in an automated way. Appending “Test" as a prefix before the name of the class, appending “Test”, “TestCase" and “TestSuite”as a suffix after the name of the class were the other two commonly used convention we have noticed and taken into account.

We calculate the rank-order correlation coefficient ρ and its significance value p of the t-value derived from each coefficient value for each source-based metric and the dLOC and dNOTC metrics of the test suite. The pairing process had been completed before measuring the statistical functions. Thus, we use PL/SQL and embedded Extended Statistical Function Set of Oracle 11g RDBMS [48] to calculate the required values. t-value is calculated finally using the correlation coefficient value and the number of pairs involved. The statistical significance (p) of t is obtained from a standard table. [32] This process is repeated for all our projects and their metrics data set.

5.4.6 Statistical Results

In this section, we present the results of the experiments we have performed on our study data. The following tables hold the results for significant projects, respectively. We also define three other project groupings, as well according to different scopes. In two of them, the projects are grouped by developer framework, i.e. Apache and JBoss, respectively. In the last grouping, all projects are considered to be one single project.

First tables of each group contain the values of Spearman's rank-order correlation coefficient (ρ) [40] for source-based metric m and both test suite metrics dLOC and dNOTC. Likewise, second tables of each group contain the statistical significance (p-value) of the t value derived from each ρ.

The detailed results and data sets used to compute the results are displayed in the following tables and in the appendices, APPENDIX D, APPENDIX E and APPENDIX F. The statistical assessments of these tables are given in the following section.

Based on these results, we evaluate hypotheses $\mathbf{H_0}$(x; y) and $\mathbf{H_1}$(x; y). They were previously defined as:

$\mathbf{H_0}$(x; y): There is no association between design metric x and test suite metric y,

$\mathbf{H_1}$(x; y): There is an association between design metric x and test suite metric y,

where x ranges over our set of source-based metrics, and y is either the dLOCC or dNOTC of associated test suites.

By definition of ρ, and correlation measures in general, if two variables are independent, i.e. there is no association between them, then ρ = 0. Thus if our results show that if $\rho(x; y) \neq 0$ for some x and y, then there is an association between x and y. In other words, if $\rho(x; y) \neq 0$, we can reject $\mathbf{H_0}$(x; y) and accept the converse, $\mathbf{H_1}$(x; y). The statistical significance p(x; y) indicates the probability that the observed value of ρ(x; y) is a chance event. Therefore, if the value of p(x; y) is low, we can confidently reject $\mathbf{H_0}$(x; y), and accept $\mathbf{H_1}$(x; y). We can reject $\mathbf{H_0}$(x; y) at a certain confidence level of α, if [7]

$$\mathbf{1 - p(x; y) < \alpha.}$$

Table 9 : Correlation Results of All Projects as One Single Project – Class Level Metrics

Correlation Coefficients ρ(x; y)	**dLOC_CLS**	**dLOC_CLS_ NEW**	**dNOTC_CLS**	**dNOTC_CLS _NEW**
DIT	0.064363897	0.107963371	0.088405433	0.108954434
FOUT	0.34682534	0.392640796	0.24496436	0.395120813
LCOM	0.269135257	0.304801531	0.080633395	0.306301052
LOC_CLS	0.420211132	0.460308528	0.293085114	0.461930551
LOC_CLS _NEW	0.42808285	0.472410942	0.296477966	0.474807542
NOF	0.315836069	0.341247827	0.093128928	0.34330938
NOM	0.365195767	0.399465272	0.189112224	0.401551696
NORM	0.108212199	0.131345198	0.109371547	0.131198522
NSC	0.085259606	0.094764236	0.085241835	0.09388952
NSF	0.158191716	0.16964695	0.117527905	0.169183456
NSM	0.041801031	0.052054527	0.16084185	0.053092041
RFC	0.378856562	0.432692488	0.291151517	0.436800301
SIX	0.065195645	0.090207252	0.086596139	0.089897277
TNOF	0.331251231	0.355577461	0.151231252	0.357659544
TNOM	0.376346157	0.416197694	0.247087729	0.41895352
WMC	0.417520789	0.46053687	0.299988817	0.4622385

Table 10 : Significance Results of All Projects as One Single Project – Class Level Metrics

Significance Values p(x; y)	**dLOC_CLS**	**dLOC_CLS_ NEW**	**dNOTC_CLS**	**dNOTC_CLS _NEW**
DIT	0.012161513	2.51013E-05	0.000566454	2.11114E-05
FOUT	4.05E-44	4.31E-57	3.63E-22	7.43E-58
LCOM	1.39E-26	5.59E-34	0.001671717	2.59E-34
LOC_CLS	5.90E-66	2.06E-80	1.97E-31	4.88E-81
LOC_CLS _NEW	1.21E-68	3.61E-85	3.71E-32	3.91E-86
NOF	1.73E-36	1.11E-42	0.000281064	3.30E-43
NOM	4.54E-49	3.29E-59	1.11331E-13	7.25E-60
NORM	2.4037E-05	2.84009E-07	1.96194E-05	2.92871E-07
NSC	0.000887151	0.000218833	0.000889366	0.000250302
NSF	5.81637E-10	2.92419E-11	4.43887E-06	3.31387E-11
NSM	0.103638491	0.042646199	2.96731E-10	0.038675868
RFC	5.76E-53	3.00E-70	5.06E-31	1.06E-71
SIX	0.011088611	0.000435253	0.000734492	0.000455589
TNOF	3.57E-40	1.95E-46	3.22963E-09	5.34E-47
TNOM	3.09E-52	1.30E-64	1.55E-22	1.56E-65
WMC	4.71E-65	1.69E-80	6.43E-33	3.70E-81

Table 11 : Correlation Results of JBoss Projects Only – Class Level Metrics

Correlation Coefficients ρ(x; y)	dLOC_CLS	dLOC_CLS_ NEW	dNOTC_CLS	dNOTC_CLS _NEW
DIT	0.205678743	0.206871759	0.078605678	0.209313114
FOUT	0.377043065	0.391392368	0.167802649	0.395685179
LCOM	0.215971529	0.238712308	0.035124348	0.239626463
LOC_CLS	0.41377393	0.435882329	0.159107246	0.438358763
LOC_CLS _NEW	0.426637315	0.450236384	0.17962364	0.453306727
NOF	0.282602288	0.305524822	0.090544726	0.306169537
NOM	0.354099107	0.373687767	0.127829245	0.374922247
NORM	0.140438068	0.142182085	0.028478126	0.141617119
NSC	0.054226891	0.042136767	-0.025683244	0.044611204
NSF	0.106623102	0.103005492	0.019648467	0.10516987
NSM	-0.054295796	-0.048645186	-0.031737553	-0.044075264
RFC	0.410581957	0.431952508	0.216598094	0.437148644
SIX	0.134963824	0.137834309	0.020175969	0.137291781
TNOF	0.279549475	0.296238005	0.081427899	0.298366487
TNOM	0.357632262	0.387037055	0.148739934	0.389099992
WMC	0.415187928	0.439967668	0.18092358	0.442056973

Table 12 : Significance Results of JBoss Projects Only – Class Level Metrics

Significance Values p(x; y)	dLOC_CLS	dLOC_CLS_ NEW	dNOTC_CLS	dNOTC_CLS _NEW
DIT	9.11291E-06	8.06223E-06	0.092910882	6.26063E-06
FOUT	6.40837E-17	3.23558E-18	0.000309845	1.28666E-18
LCOM	3.09332E-06	2.3456E-07	0.453332316	2.1027E-07
LOC_CLS	2.27229E-20	1.16E-22	0.000631913	6.26E-23
LOC_CLS _NEW	1.10511E-21	3.04E-24	0.000111048	1.37E-24
NOF	7.39582E-10	2.37219E-11	0.052815581	2.14384E-11
NOM	5.63429E-15	1.26138E-16	0.006154465	9.841E-17
NORM	0.002593323	0.002288233	0.543242953	0.002383275
NSC	0.246787673	0.368279085	0.583531336	0.340799464
NSF	0.022483	0.027506347	0.674933346	0.024396418
NSM	0.246186855	0.298887219	0.498071065	0.346637403
RFC	4.71652E-20	3.05E-22	2.89141E-06	8.47E-23
SIX	0.003807202	0.003118164	0.666723969	0.003238992
TNOF	1.14282E-09	9.91819E-11	0.081725161	7.17781E-11
TNOM	2.89441E-15	8.13434E-18	0.001411693	5.26534E-18
WMC	1.64012E-20	4.19E-23	9.87958E-05	2.47E-23

Table 13 : Correlation Results of Apache Projects Only – Class Level Metrics

Correlation Coefficients ρ(x; y)	**dLOC_CLS**	**dLOC_CLS_ NEW**	**dNOTC_CLS**	**dNOTC_CLS _NEW**
DIT	0.012412594	0.073447423	0.10749292	0.074510185
FOUT	0.319453739	0.382687884	0.273075098	0.38395898
LCOM	0.288510929	0.330410318	0.09317706	0.331779679
LOC_CLS	0.416325751	0.464755091	0.353277397	0.465518049
LOC_CLS _NEW	0.421699223	0.476016244	0.34788689	0.477695198
NOF	0.335302812	0.359919937	0.09319421	0.362345077
NOM	0.35683461	0.399393072	0.203292438	0.401870569
NORM	0.069025426	0.106943288	0.139583877	0.106861951
NSC	0.097074269	0.116060012	0.139524561	0.114115825
NSF	0.176812782	0.196919947	0.157735603	0.195103657
NSM	0.076929714	0.090799339	0.248227882	0.090110061
RFC	0.352738184	0.423864068	0.317201876	0.427220454
SIX	0.0237431	0.060764704	0.116132898	0.060722786
TNOF	0.35750432	0.384318851	0.18290689	0.38594002
TNOM	0.374849124	0.419919724	0.284709336	0.422798967
WMC	0.409589977	0.461542444	0.350770198	0.462763534

Table 14 : Significance Results of Apache Projects Only – Class Level Metrics

Significance Values p(x; y)	**dLOC_CLS**	**dLOC_CLS_ NEW**	**dNOTC_CLS**	**dNOTC_CLS _NEW**
DIT	0.686600571	0.016823011	0.000458169	0.015298288
FOUT	1.50E-26	2.88E-38	1.44926E-19	1.57E-38
LCOM	9.50E-22	2.16E-28	0.002403617	1.25E-28
LOC_CLS	1.20E-45	7.38E-58	1.74E-32	4.56E-58
LOC_CLS _NEW	6.61E-47	5.43E-61	1.73E-31	1.81E-61
NOF	3.07E-29	9.72E-34	0.002399149	3.33E-34
NOM	3.75E-33	7.95E-42	2.42828E-11	2.26E-42
NORM	0.024686623	0.000490173	5.1312E-06	0.000495082
NSC	0.001562814	0.000153588	5.17922E-06	0.000198148
NSF	6.9225E-09	1.01937E-10	2.47816E-07	1.52103E-10
NSM	0.012272047	0.003102063	2.46486E-16	0.003336594
RFC	2.20E-32	2.02E-47	3.50E-26	3.18E-48
SIX	0.440202733	0.048051378	0.000152117	0.048206053
TNOF	2.80E-33	1.32E-38	2.02452E-09	6.06E-39
TNOM	1.15E-36	1.74E-46	3.37503E-21	3.63E-47
WMC	4.22E-44	5.50E-57	5.09E-32	2.57E-57

5.4.7 Statistical Assessment

Examining the correlation and significance result tables given above, it is seen that for relations among some metrics, $\mathbf{H_0}$(x; y) can be rejected and $\mathbf{H_1}$(x; y) can be accepted at the 99% level of confidence, whereas some relations have 95% level of confidence. Some metrics are significantly correlated with one of the test suite metrics if the confidence level is lowered to 95%. Thus, we have tried to use the most suitable level of confidence according to the correlation values.

In addition to calculating correlations among the source and test metrics, we also calculated the correlations among the source-based metrics themselves. These correlations are discussed in the related sections below, and the details of correlation analysis are given in APPENDIX A.

Examining the correlations among the source-based metrics, we observe that many of the source-based metrics are correlated among each other. For the three cases (single project, Apache and JBoss separately), they seem to have similar groups of metrics that are all strongly and moderately correlated to each other. The detailed comments on the correlation among the metrics are given in the following analysis results under each metric respectively.

5.4.7.1 Method-Level Metrics

Below are given the results of correlation analysis we have performed among the four method-level source metrics for the source classes. Significance values of the correlation results for all three cases are omitted, as they all have significant correlation values and high levels of confidence with a significance value of 0 for all measurements.

The correlation analysis indicates that the lines of code in the methods is strongly related to the McCabe Cyclomatic Complexity (VG) [37] metric of the method, as one would normally expect size and complexity to be correlated strongly. VG defines complexity as it is a measure of the complexity of a modules decision structure.

Nested Block Depth (NBD) metric is an indicator of complex control flow within the program. Deeply nested conditional statements increase the conceptual complexity of the code and are more likely to be error-prone. Therefore, it is normal to see a strong correlation between this metric and the Method Lines of Code (MLOC), as deeply nested conditional statements means increase in size of the code.

Number of Parameters (PAR) metric measures the number of parameters that are passed to a method, and it seems to have a moderate correlation with the other three methods metrics. On the other hand, VG metric is tightly correlated to the NBD metric with a value of 0.8914. This may result from the fact that both of these metrics aim to measure the same design parameter, complexity of the method, from different points of view.

All three cases (single project, Apache and JBoss separately) exhibit very similar patterns for the six metric correlation measurements. The correlations have differed negligibly, meaning that the relation among the methods metrics we have used in our study are independent of the context the software is developed and we can make a generalization with these results.

Table 15 : Correlation Values Among Method Metrics of Source Classes–All as One Single Project

	MLOC	*NBD*	*PAR*	*VG*
MLOC	1			
NBD	0.7818	1		
PAR	0.2797	0.3111	1	
VG	0.7892	0.8914	0.3232	1

Table 16 : Correlation Values Among Method Metrics of Source Classes–Apache Projects Only

	MLOC	*NBD*	*PAR*	*VG*
MLOC	1			
NBD	0.7778	1		
PAR	0.2954	0.3131	1	
VG	0.7923	0.9026	0.3197	1

Table 17 : Correlation Values Among Method Metrics of Source Classes – JBoss Projects Only

	MLOC	*NBD*	*PAR*	*VG*
MLOC	1			
NBD	0.7854	1		
PAR	0.2584	0.3061	1	
VG	0.7843	0.8762	0.3264	1

The results of correlation analysis we have performed among the four method level source metrics for the test classes are listed in Tables 17, 18 and 19. The significance values of the correlation results for all three cases are shown in the same tables. Most of the correlation analysis have significant and high levels of confidence correlation with a significance value of 0 for all measurements. Only JBoss-projects-only case has two measurements one of which is acceptable, i.e. 0.05 whereas the other value (0.152) is too high to accept.

The correlation analysis indicates that the lines of code in the methods has a strongly moderate correlation to the McCabe Cyclomatic Complexity (VG) [37] metric of the method with a value less than the source class value, as one would normally expect size and complexity to be correlated strongly.

Due to the same reasons stated for the source class analysis, it is normal to see a moderate-strong correlation between NBD metric and the Method Lines of Code (MLOC), as deeply nested conditional statements means increase in size of the code. Again, we note that, the correlation value is weaker than the source class correlation value.

Number of Parameters (PAR) metric seems to have a moderate correlation with the MLOC metric and weak correlations with the other two methods metrics. This pattern is different from the source class correlation analysis. On the other hand, VG metric is tightly correlated to the NBD metric with a value close to 0.90. This may also result from the fact that both of these metrics aim to measure the complexity degree of the test methods, from different points of view.

All three cases (single project, Apache and JBoss separately) exhibit very similar patterns for the six metric correlation measurements. Different from the source class correlation analysis, the correlations differ significantly for PAR metric. However, for the other metrics and their cross correlation measurement, correlation values imply that the relations among the methods metrics we have used in our study are independent of the context the software is developed and we can make a generalization with these results, as the values are nearly the same for three cases.

Table 18 : Correlation Analysis Among Methods Metrics of Test Classes: All As One Single Project

Correlation	*MLOC*	*NBD*	*PAR*	*VG*
MLOC	1			
NBD	0.554	1		
PAR	-0.228	0.067	1	
VG	0.536	0.911	0.088	1

Significance	*MLOC*	*NBD*	*PAR*	*VG*
MLOC	0			
NBD	0	0		
PAR	0	6.5E-41	0	
VG	0	0	1.1E-69	0

Table 19 : Correlation Analysis Among Methods Metrics of Test Classes : Apache Projects Only

Correlation	*MLOC*	*NBD*	*PAR*	*VG*
MLOC	1			
NBD	0.542	1		
PAR	-0.143	0.130	1	
VG	0.529	0.911	0.146	1

Significance	*MLOC*	*NBD*	*PAR*	*VG*
MLOC	0			
NBD	0	0		
PAR	1E-102	2E-84	0	
VG	0	0	2E-107	0

Table 20 : Correlation Analysis Among Methods Metrics of Test Classes : JBoss Projects Only

Correlation	*MLOC*	*NBD*	*PAR*	*VG*
MLOC	1			
NBD	0.569	1		
PAR	-0.333	-0.014	1	
VG	0.546	0.910	0.010	1

Significance	*MLOC*	*NBD*	*PAR*	*VG*
MLOC	0			
NBD	0	0		
PAR	0	0.050	0	
VG	0	0	0.152	0

5.4.7.2 Class-Level Metrics

One of the most important consequences we can obtain from the correlation values with the test metrics is that, the new metrics we have proposed using the correlation coefficients of class metrics with dNOTC_CLS and dLOC_CLS metrics, i.e. dNOTC_CLS_NEW and dLOC_CLS_NEW have nearly the same correlation values with the class metrics we have used in our assessments. This results from the statistical fact that dNOTC_CLS and dLOC_CLS are strongly correlated with a correlation coefficient of 0.99. Therefore, instead of two metrics, only one of the new metrics proposed is enough to investigate the relations of these new metrics with the class metrics. Their significance values are also very close to each other providing very high levels of confidence for most of the measurements.

In the following section, we will try to identify the relationship between each source-based class metrics and test suite metrics, so that we can understand how the design parameter the related metric is connected to is affected to testing effort and testability.

In order to obtain a regression analysis between the class-level source-based metrics and test metrics, we will define which metrics will be included in the analysis, according to the correlation and significance values of the correlation analysis we have performed in the previous section. While choosing the metrics to be included in the analysis, we will take into account both in; case level, i.e. single project case, Apache and JBoss cases separately, results and independent project-based results all together.

The results of correlation analysis we have performed among the class level metrics are listed in Table 20 and 21. Before proceeding, we have to state that, according to the correlation analysis we have performed among all package metrics, test metrics have resulted to have moderately strong and strong correlations among each other, with very high levels of confidence, 99%. Table 20 summarizes the correlation analysis we have performed among the old and new test metrics. The significance values are omitted as they are very close to 0, satisfying a very high level of confidence, 99%.

Table 21 : Correlation Values Among Test Class Metrics: Correlation Analysis – All as One Single Project

	dLOC_CLS	dLOC_CLS _NEW	dNOTC_CLS	dNOTC_CLS _NEW
dLOC_CLS	1			
dLOC_CLS _NEW	0.9736	1		
dNOTC_CLS	0.6468	0.6123	1	
dNOTC_CLS _NEW	0.9757	0.9994	0.9648	1

Table 22 : Correlation Values Among Class Metrics : Correlation Analysis – All as One Single Project

	DIT	***FOUT***	***LCOM***	***LOC_CLS***	***NOF***	***NOM***	***NORM***	***NSC***	***NSF***	***NSM***	***RFC***	***SIX***	***TNOF***	***TNOM***	***LOC_CLS _NEW***	***WMC***
DIT	1.00															
FOUT	0.09	1.00														
LCOM	-0.05	0.30	1.00													
LOC_CLS	0.07	0.81	0.44	1.00												
NOF	-0.05	0.35	0.85	0.48	1.00											
NOM	0.08	0.41	0.64	0.62	0.67	1.00										
NORM	0.31	0.21	0.18	0.22	0.22	0.33	1.00									
NSC	0.06	0.08	0.10	0.10	0.10	0.18	0.06	1.00								
NSF	0.02	0.32	0.14	0.32	0.15	0.18	0.09	0.04	1.00							
NSM	-0.03	0.30	-0.06	0.27	-0.09	-0.11	-0.04	-0.02	0.25	1.00						
RFC	0.14	0.83	0.48	0.87	0.51	0.72	0.29	0.12	0.29	0.24	1.00					
SIX	0.34	0.18	0.14	0.18	0.18	0.27	0.98	0.05	0.07	-0.05	0.25	1.00				
TNOF	-0.06	0.41	0.75	0.54	0.85	0.60	0.20	0.09	0.55	0.06	0.54	0.16	1.00			
TNOM	0.04	0.50	0.61	0.71	0.62	0.93	0.30	0.17	0.25	0.14	0.81	0.24	0.62	1.00		
LOC_CLS_NEW	0.07	0.79	0.52	0.97	0.56	0.74	0.26	0.12	0.34	0.24	0.93	0.21	0.62	0.83	1.00	
WMC	0.01	0.69	0.55	0.89	0.58	0.80	0.27	0.15	0.32	0.21	0.88	0.22	0.62	0.89	0.95	1.00

Depth Of Inheritance Tree (DIT) Metric

DIT metric is one of the few metrics that represent relatively small correlation values with the test metrics. Inheritance is one of the key design parameters that affect the amount of testing effort. The number of required test cases depends on usage of inheritance mechanism in the class and object hierarchy and the testing criterion of the project. As a class may inherit methods of other classes via inheritance mechanism, the testing criterion defines where to test these methods inherited into a class.

Therefore, it is normal to observe a correlation between DIT metric and test metrics. However, one would expect to see higher correlation values, as this metric represents a major design characteristic. It is seen from the correlation results that JBoss projects possess higher correlation with test metrics with respect to the Apache projects. This indicates a major design difference, as the two cases own their own set of rules, i.e. inheritance mechanisms, for software development.

Significance values show that DIT metric has higher level of confidence with new test metrics proposed in this study than the old test metrics. It is also seen that, NOTC test metric has a lower level of confidence, close to 90%, which may be unacceptable from statistical point of view for some projects.

The new proposed test metrics have higher correlation values and levels of confidence (meaning lower significance values) with respect to their corresponding old metric definitions. When dLOC_CLS metric has a correlation value of 0.06, dLOC_CLS_NEW metric has a value of 0.10. Likewise, when dNOTC metric has a correlation value of 0.08, dLOC_CLS_NEW metric has a value of 0.10.

For JBoss and Apache projects separately, the new metrics proposed have different effects with respect to the values of the older metrics with the DIT metric. JBoss projects show a correlation coefficient of 0.20 for both dLOC_CLS and dLOC_CLS_NEW metrics with no visible change, whereas Apache projects show a correlation coefficient of 0.012 for dLOC_CLS and 0.07 for dLOC_CLS_NEW, meaning visible change in magnitude but still negligible correlation value.

For the other test metrics, JBoss projects show a correlation coefficient of 0.07 for dNOTC and 0.20 for dNOTC_NEW, meaning visible change in magnitude and also much better correlation value to take into consideration in regression analysis. For the Apache projects, the correlation value decline for the new metric with respect to the older one. However, the value is still weak.

Examining the Apache and JBoss projects one by one, we observe that Apache Ant and Geronimo exhibit moderate correlation values all metrics, and JBoss Drools is the only project that exhibits strongly moderate correlation values with all metrics.

According to the common trend, DIT metric is included in the regression analysis although it has relatively weak correlation in general, and moderate results in a few projects with both test metrics. The inter-metrics correlation table (Table 22) also indicates that DIT metric is moderately related to NORM and SIX metrics only, and has weak correlation with the rest of the class metrics.

Fan Out (FOUT) Metric

FOUT metric is one of the metrics that represent moderate correlation values with the test metrics. Coupling is one of the key design parameters that affect the amount of testing effort. The number of required test cases depends on usage of coupling mechanism in the class and object hierarchy and the testing criterion of the project.

When a class has high coupling, this mean you have to consume more resource, both time and effort, to be able to understand and test it, as you have to trace all the coupled external pieces (other coupled classes) to obtain the functionality roadmap of the class to be tested. Besides, high coupling decreases the possibility of reusability, as the components (classes or subsystems) you want to reuse will be dependent on many other components and it will be difficult to extract the required component from its context.

Therefore, it is normal to observe a moderate correlation between FOUT metric and test metrics. It is seen from the correlation results that JBoss projects possess higher correlation with LOCC test metrics, and Apache projects possess higher correlation with NOTC test metrics. The significance values show that FOUT metric has very high levels of confidence with all test metrics, over 99%, meaning significant results.

The new proposed test metrics have higher correlation values and levels of confidence (meaning lower significance values) with respect to their corresponding old metric definitions. When dLOC_CLS metric has a correlation value of 0.34, dLOC_CLS_NEW metric has a value of 0.39. Likewise, when dNOTC metric has a correlation value of 0.24, dLOC_CLS_NEW metric has a value of 0.39.

For both JBoss and Apache projects, and considering all projects as a single project case, the new metrics proposed have similar effects with respect to the values of the older metrics with the FOUT metric. The difference between dLOC_CLS and dLOC_CLS_NEW metrics is negligible with respect to the change between dNOTC and dNOTC_NEW metrics.

Apache projects show a better correlation between FOUT and dNOTC metric with respect to JBoss projects, i.e. 0.27 versus 0.16 respectively. The critical point to note is that both JBoss and Apache projects show very close correlations between FOUT and LOC_CLS test metrics and nearly the same correlations between FOUT and NOTC test metrics. This may indicate that coupling was handled in a similar manner for both software communities, although they own their own set of rules, i.e. coupling mechanisms, for software development.

The correlation results also showed that FOUT is a significantly better predictor of the dLOC_CLS metric than of the dNOTC metric. Thus, the association between the fan out of a class and the size of its test suite is significantly stronger than the association between the fan out and the number of test cases.

The fan out of a class measures the number of other classes that the class depends on. At the run-time, these classes will have to be initialized, and the fields of the classes will be set to the appropriate values before they are used. When a class needs to be (unit) tested, however, the tester will need to take care of the initialization of the (objects of) other classes and the class-under-test itself. The amount of initialization required before testing can be done will thus influence the testing effort, and by assumption, the dLOC_CLS metric. [7]

Examining the Apache and JBoss projects one by one, we observe that Apache Ant, Geronimo, Mina and JBoss Drools exhibit strongly moderate correlation values with all test metrics except for dNOTC metric and moderate correlation values with dNOTC test metrics. Apache Lucene, Wicket and JBoss Richfaces exhibit moderate correlation values with all test metrics.

In summary, both dLOCC and dNOTC test metrics are moderately correlated to FOUT metric. Results show that FOUT and dLOC_CLS test metrics correlated more than dNOTC metric correlations. As dLOC_CLS and dNOTC metrics are correlated with each other as a natural consequence between the size of the software and number of test cases required to test the software, it is normal to have moderate correlations between FOUT and test metrics.

According to the common trend, FOUT metric is included in the regression analysis as it has significant and promising correlation with both test metrics. The inter-metrics correlation table (Table 22) also indicates that FOUT metric is strongly correlated to LOC_CLS, RFC, WMC, LOC_CLS_NEW and NOTC_NEW. This metric has moderate correlation with most of the metrics, too.

Lack of Cohesion of Methods (LCOM) Metric

LCOM metric is one of the metrics that represent moderate correlation values with the LOC_CLS test metrics, but weak values with the NOTC test metric.

Cohesiveness of methods within a class is desirable, since it promotes encapsulation. Lack of cohesion implies classes should probably be split into two or more subclasses. Any measure of disparateness of methods helps identify flaws in the design of classes. Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.

Therefore, it is normal to observe a moderate correlation between LCOM metric and new proposed test metrics, but interesting to observe a weak relation with dNOTC metric. It is seen from the correlation results that Apache projects possess slightly higher correlation than JBoss projects with test metrics, meaning that their cohesion handling rules may be close to each other. The significance values show that LCOM metric has very high levels of confidence with all test metrics, over 99%, meaning significant results.

The new proposed test metrics have higher correlation values and levels of confidence (meaning lower significance values) with respect to their corresponding old metric definitions. When dLOC_CLS metric has a correlation value of 0.28, dLOC_CLS_NEW metric has a value of 0.33. Likewise, when dNOTC metric has a correlation value of 0.09, dLOC_CLS_NEW metric has a value of 0.33.

For both JBoss and Apache projects, and considering all projects as a single project case, the new metrics proposed have similar effects with respect to the values of the older metrics with the LCOM metric. The difference between dLOC_CLS and dLOC_CLS_NEW metrics is negligible with respect to the change between dNOTC and dNOTC_NEW metrics.

The correlation results also showed that LCOM is a significantly better predictor of the dLOC_CLS metric than of the dNOTC metric. Thus, the association between the lack of cohesion of methods out of a class and the size of its test suite is significantly stronger than the association between the lack of cohesion of methods and the number of test cases.

Examining the Apache and JBoss projects one by one, we observe that Apache Ant and Geronimo exhibit moderate correlation values with all metrics except for dNOTC metric, moderately weak correlation with dNOTC metrics. Apache Lucene and Wicket exhibit moderate correlation values all metrics.

According to the common trend, LCOM metric is included in the regression analysis as it exhibits moderate results in general. The inter-metrics correlation table (Table 22) also indicates that LCOM metric is strongly correlated to NOF, NOM, TNOF, TNOM and WMC. This metric has moderate correlation with most of the metrics, too.

Lines Of Code for Class (LOC_CLS & LOC_CLS_NEW) Metrics

LOC_CLS metrics are the metrics that represent highest moderately strong correlation values with the LOC_CLS test metrics, but moderate values with the NOTC test metric, in the case of considering all projects as one single project. The results differ for Apache and JBoss projects cases.

A common basis of estimate on a software project is LOC_CLS metrics. LOC_CLS are used to create time and cost estimates. The LOC_CLS estimate becomes the baseline to measure the degree of work performed on a project. Once a project is underway, the LOC_CLS becomes a tracking tool to measure the degree of progress on a module or project. An experienced developer can gage a LOC_CLS estimate based upon knowledge of past productivity on projects. The LOC_CLS measurement becomes the barometer for the program's progress and productivity.

LOC_CLS metrics represent highest moderately strong correlation values with both LOC_CLS and NOTC test metrics for Apache projects. However, for JBoss projects, there seems to have moderate values with the NOTC test metric, which degrades the correlation values with the NOTC test metric in the case of considering all projects as one single project.

The JBoss NOTC results are surprising, as it is normal to expect a better correlation between LOC_CLS metrics and NOTC metric. Normally, one expects a larger class to have more test cases in corresponding test class, as the size of the source class means that it contains more functionality inside with respect to small-size classes. Examining the JBoss project-based correlation results as given in the APPENDIX D, it is seen that the problem is caused by one of the projects by JBoss taken into consideration, which is JBoss Cache project. After detailed analysis of the source code, it was seen that not all of the test suites and cases were loaded into the version we have used. Thus, the problem with the correlation values between LOCC metrics and dNOTC test metric is ignored, as the other two projects show even better correlation values than the results of the case of considering all projects as one single project.

The new proposed test metrics have higher correlation values and levels of confidence (meaning lower significance values) with respect to their corresponding old metric definitions. Considering LOC_CLS metric, when dLOC_CLS metric has a correlation value of 0.42, dLOC_CLS_NEW metric has a value of 0.46. Likewise, when dNOTC metric has a correlation value of 0.29, dLOC_CLS_NEW metric has a value of 0.46. Considering LOC_CLS_NEW metric, when dLOC_CLS metric has a correlation value of 0.42, dLOC_CLS_NEW metric has a value of 0.47. Likewise, when dNOTC metric has a correlation value of 0.29, dLOC_CLS_NEW metric has a value of 0.47.

The correlation results also showed that LOC_CLS is a significantly better predictor of the dLOCC metric than of the dNOTC metric. Thus, the association between the line count of a class and the size of its test suite is significantly stronger than the association between the line count and the number of test cases.

Examining the Apache and JBoss projects one by one, we observe that Apache Ant and Geronimo exhibit moderately strong correlation values with all metrics except for dNOTC metric, moderate correlation with dNOTC metric. Apache Lucene, Mina and JBoss Drools exhibit moderately strong correlation values all metrics. Apache Wicket and JBoss Richfaces exhibit moderate correlation values all metrics.

According to the common trend, LOC_CLS metric is included in the regression analysis as it exhibits moderately strong and moderate results in general. The inter-metrics correlation table (Table 22) also indicates that LOCC metric is strongly correlated to FOUT, NOM, RFC, TNOM, WMC, LOC_CLS_NEW and NOTC_NEW. This metric has moderate correlation with most of the metrics, too.

Number of Attributes (NOF) and Total Number of Fields (TNOF) Metrics

NOF metric is one of the other metrics that represent moderate correlation values with the LOC_CLS test metrics, but weak values with the NOTC test metric. The attributes (fields) of the class-under-test need to be initialized before testing can be done. This means that the amount of required initialization affects the testing effort and the dLOC_CLS metric. Thus, we expect correlation between the NOF and dLOC_CLS metrics, which is just the case in the results.

For all three cases (single project, Apache and JBoss separately), the results show the same pattern between the correlation values with dLOC_CLS and dNOTC metric as stated above. The interesting result seen from the tables is that all three situations have the same correlation values between NOF and dNOTC metric, a value of 0.09. The significance values differ for only JBoss projects, as 95% is the necessary level of confidence for this situation, whereas 99% is the value for the other two cases.

The new proposed test metrics have higher correlation values and levels of confidence (meaning lower significance values) with respect to their corresponding old metric definitions. When dLOC_CLS metric has a correlation value of 0.28, dLOC_CLS_NEW metric has a value of 0.31. Likewise, when dNOTC metric has a correlation value of 0.09, dLOC_CLS_NEW metric has a value of 0.31.

For both JBoss and Apache projects, and considering all projects as a single project case, the new metrics proposed have similar effects with respect to the values of the older metrics with the NOF metric. The difference between dLOC_CLS and dLOC_CLS_NEW metrics is negligible with respect to the change between dNOTC and dNOTC_NEW metrics.

The correlation results also showed that NOF is a significantly better predictor of the dLOC_CLS metric than of the dNOTC metric. Thus, the association between the number of attributes of a class and the size of its test suite is significantly stronger than the association between the number of attributes and the number of test cases.

Examining the Apache and JBoss projects one by one, we observe that Apache Ant, Lucene, Geronimo show moderately strong correlation values with the LOC_CLS test metrics. Apache Lucene is the only project that exhibits moderately strong correlation values with the NOTC test metrics.

The Total Number of Fields (TNOF) metric follow exactly the same statistical pattern and results with Number of Attributes (NOF) metric, therefore, its statistical assessment is the same as above and so not stated separately. The reason lies in the fact that NOF and TNOF metrics have a strong correlation between each other. (0.85 in magnitude)

According to the common trend, NOF and TNOF metrics are included in the regression analysis as they exhibit moderate results in general. The inter-metrics correlation table (Table 22) also indicates that NOF and TNOF metrics are moderately-strongly correlated to LCOM, RFC, NOM, TNOM, WMC, LOC_CLS_NEW and NOTC_NEW. These metrics have weak correlation with the rest of the metrics.

<u>Number of Methods (NOM) and Total Number of Methods (TNOM) Metrics</u>

NOM metric is one of the other metrics that represent moderate correlation values with the LOCC test metrics, but weak values with the NOTC test metric.

For all three cases (single project, Apache and JBoss separately), the results show the same pattern between the correlation values with dLOC_CLS and dNOTC metric as stated above. The interesting result seen from the tables is that all three situations have very close correlation values between NOM and dLOC_CLS metric, around a value of 0.36.

The new proposed test metrics have higher correlation values and levels of confidence (meaning lower significance values) with respect to their corresponding old metric definitions. When dLOC_CLS metric has a correlation value of 0.36, dLOC_CLS_NEW metric has a value of 0.40. Likewise, when dNOTC metric has a correlation value of 0.19, dLOC_CLS_NEW metric has a value of 0.40.

For both JBoss and Apache projects, and considering all projects as a single project case, the new metrics proposed have similar effects with respect to the values of the older metrics with the NOM metric. The difference between dLOC_CLS and dLOC_CLS_NEW metrics is negligible with respect to the change between dNOTC and dNOTC_NEW metrics.

The correlation results also showed that NOM is a significantly better predictor of the dLOC_CLS metric than of the dNOTC metric. Thus, the association between the number of methods of a class and the size of its test suite is significantly stronger than the association between the number of methods and the number of test cases.

Examining the Apache and JBoss projects one by one, we observe that Apache Ant, Lucene, Mina and JBoss Drools show strong correlation values with the LOC_CLS test metrics. Apache Ant, Lucene, Mina, JBoss Drools are the projects that exhibits moderate and moderately strong correlation values with the NOTC test metrics.

The Total Number of Methods (TNOM) metric follow exactly the same statistical pattern and results with Number of Methods (NOM) metric, therefore, its statistical assessment is the same as above and so not stated separately.

According to the common trend, NOM and TNOM metrics are included in the regression analysis as they exhibit moderate results in general. The inter-metrics correlation table (Table 22) also indicates that NOM and TNOF metrics are moderately-strongly correlated to LCOM, LOC_CLS, RFC, NOF, TNOF, WMC, LOC_CLS_NEW and NOTC_NEW. These metrics have weak correlation with the rest of the metrics.

Number of Overridden Methods (NORM) Metric

NORM metric is one of the other metrics that represent weak-moderate correlation values both with the LOCC and NOTC test metrics.

The number of redefined operations plays a role in the specialization of the class and must be maintained in a proportion that continues to justify inheritance. Too many redefined operations imply too big a difference with the parent class and inheritance then makes less sense.

The new proposed test metrics have slightly higher correlation values and levels of confidence (meaning lower significance values) with respect to their corresponding old metric definitions. When dLOC_CLS metric has a correlation value of 0.10, dLOC_CLS_NEW metric has a value of 0.13. Likewise, when dNOTC metric has a correlation value of 0.10, dLOC_CLS_NEW metric has a value of 0.13.

The correlation results also showed that NORM is a significantly better predictor of the dLOC_CLS metric than of the dNOTC metric. Thus, the association between the number of overridden methods of a class and the size of its test suite is significantly stronger than the association between the number of overridden methods and the number of test cases.

Examining the Apache and JBoss projects one by one, we observe that JBoss Drools is the only the project that exhibits moderate correlation values with the LOC_CLS test metrics. All projects follow the same pattern as in the single project case for NOTC test metrics.

According to the common trend, NORM metric is partially, i.e. for some test metrics, included in the regression analysis as they exhibit moderate results in general. The inter-metrics correlation table (Table 22) also indicates that NORM metric is surprisingly strongly correlated to SIX metric only with a value of 0.98, and has weak and moderate correlation with most of the metrics.

<u>**Number of Children (NSC) Metric**</u>

NSC metric is one of the other metrics that represent weak correlation values both with the LOC_CLS and NOTC test metrics.

NSC measures the breadth of a class hierarchy, where maximum DIT measures the depth. Depth is generally better than breadth, since it promotes reuse of methods through inheritance. High NSC has been found to indicate fewer faults. This may be due to high reuse, which is desired.

High NSC indicates high reuse, since inheritance is a form of reuse. A large number of children (high NSC) may also mean improper abstraction of the parent class. If a class has too many children, it may indicate misuse of sub-classing. A class with many children may also require more testing.

JBoss projects have insignificant correlations, and thus ignored in community-based analysis. For all three cases (single project, Apache and JBoss separately), the results show the same pattern between the correlation values with dLOC_CLS and dNOTC metric as stated above. The interesting result seen from the tables is that all three situations have the nearly same correlation values between class and test metrics, a value of 0.09. The significance value is 99% for all three cases. It is not a common situation that the new proposed test metrics have nearly the same correlation values and levels of confidence (meaning lower significance values) with respect to their corresponding old metric definitions.

The correlation results also showed that NSC is not a good predictor of the test metrics. Thus, the association between the number of children of a class and the size of its test suite is negligible in our regression analysis.

The result change insignificantly for JBoss and Apache projects only. Examining the

Apache and JBoss projects one by one, we observe that Apache Ant, Mina, JBoss Cache exhibits moderate correlation values with the dLOC_CLS, dLOC_CLS_NEW and dNOTC_NEW test metrics. All projects follow the same pattern as in the single project case for dNOTC test metric.

The NSC metric has negligible, in other words, no significant association with all test suite metric for both systems. In the context of unit testing at the class level, the number of child classes of a class seems of little relevance to the testability.

First, the child classes are tested by their own test classes. Second, any other effects of having child classes (polymorphism) are not of concern during testing of the parent class, but during the testing of classes that use the parent class. Objects of the child classes could be used instead of objects of the parent class, possibly requiring more testing. In any case, such effects lie outside of the scope of this dissertation, as the factors of a class that influence the required testing effort for that same class is our primary concern. [7]

According to the common trend, NSC is not included in the regression analysis as it exhibits weak results in general. The inter-metrics correlation table (Table 22) also indicates that NSC metric is surprisingly weakly correlated to all metrics.

Number of Static Attributes (NSF) Metric

NSF metric is one of the other metrics that represent weak-moderate correlation values both with the LOC_CLS and NOTC test metrics. Raising the number of static attributes translates into memory footprint increase and more complexity on the application.

The new proposed test metrics have very close correlation values and levels of confidence (meaning lower significance values) with respect to their corresponding old metric definitions. When dLOC_CLS metric has, a correlation value of 0.15, dLOC_CLS_NEW metric has a value of 0.17. Likewise, when dNOTC metric has a correlation value of 0.12, dLOC_CLS_NEW metric has a value of 0.17.

For both JBoss and Apache projects, and considering all projects as a single project case, the new metrics proposed have similar effects with respect to the values of the older metrics with the NSF metric. The difference between dLOC_CLS and dLOC_CLS_NEW metrics is negligible with respect to the change between dNOTC and dNOTC_NEW metrics.

Examining the Apache and JBoss projects one by one, we observe that Apache Ant, Lucene, Geronimo, Mina, JBoss Cache exhibits moderate correlation values with the dLOC_CLS, dLOC_CLS_NEW and dNOTC_NEW test metrics, but the levels of confidence fall down to 95% for most of the projects. All projects, except for JBoss Drools follow the same pattern as in the single project case for dNOTC test metric. JBoss Drools is the only project that exhibits moderate correlation values with the all test metrics.

According to the common trend, NSF metric is included in the regression analysis as it exhibits moderate results in general. The inter-metrics correlation table (Table 22) also indicates that NSF metric has a strongly moderate correlation with TNOF metric and it exhibits moderate correlation with most of the metrics.

Number of Static Methods (NSM) Metric

Static calls are faster than dynamic ones, translating into a performance increase. However, the abuse of static methods leads to a brittle solution that does not improve the reuse factor.

NSM metric is one of the other metrics that represent weak correlation values with the dLOC_CLS, dLOC_CLS_NEW and dNOTC_NEW test metrics, but moderate correlation with dNOTC metric. JBoss projects have insignificant correlations, and thus ignored in community-based analysis. Apache project have even higher correlation values between NSM and dNOTC metric.

The new proposed test metrics have different behaviors with respect to their corresponding old metric definitions. When dLOC_CLS metric has, a correlation value of 0.04, dLOC_CLS_NEW metric has a value of 0.15, which means a negligible increase. On the other side, when dNOTC metric has a correlation value of 0.16, dLOC_CLS_NEW metric has a value of 0.05, which is a dramatic fall.

Examining the Apache and JBoss projects one by one, we observe that Apache Ant and Geronimo, exhibit moderate correlation values with all test metrics, different than the case considering all projects as a single project, but the levels of confidence fall down to 94% for Apache Geronimo, but it is still 99% for Apache Ant. All projects, except for JBoss Drools follow the same pattern as in the single project case for dNOTC test metric. JBoss Drools is the only project that exhibits moderate correlation values with the all test metrics.

According to the common trend, NSM metric is not included in the regression analysis as it exhibits weak results in general. The inter-metrics correlation table (Table 22) also indicates that NSM metric has a weak-moderate correlation with most of the metrics.

Response For Class (RFC) Metric

RFC metric is one of the metrics that represent moderately strong correlation values with the dLOC_CLS, dLOC_CLS_NEW and dNOTC_NEW test metrics, but moderate values with the dNOTC test metric, for all three cases (single project, Apache and JBoss separately).

If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding on the part of the tester. The larger the number of methods that can be invoked from a class, the greater the complexity of the class. A worst-case value for possible responses will assist in appropriate allocation of testing time.

RFC of c is a count of the number of methods of a class and the number of methods of other classes that are potentially called by the methods of this class. From the definition, it is clear that the RFC metric consists of two components. First, the number of methods of the class. The strong correlation between the RFC and NOM metrics for both systems is explained by this component. Second, the number of methods of other classes that are potentially invoked by the methods of the class. The invocation of methods of other classes gives rise to fan out, hence the strong correlation between RFC and FOUT in both systems. Given the correlations between the RFC metric and both the NOM and FOUT metrics, the observed correlations between the RFC and dLOCC metrics are as expected. [7]

The new proposed test metrics have higher correlation values and levels of confidence (meaning lower significance values) with respect to their corresponding old metric definitions. When dLOC_CLS metric has, a correlation value of 0.38, dLOC_CLS_NEW metric has a value of 0.43. Likewise, when dNOTC metric has a correlation value of 0.29, dLOC_CLS_NEW metric has a value of 0.43.

For JBoss and Apache projects separately, the new metrics proposed have different effects with respect to the values of the older metrics with the RFC metric. Apache projects show the same pattern as stated above, but for JBoss projects, the change in LOC_CLS test metrics is negligible.

Examining the Apache and JBoss projects one by one, we observe that Apache Ant, Geronimo and JBoss Cache exhibit strong correlation values with the dLOC_CLS, dLOC_CLS_NEW and dNOTC_NEW test metrics, and moderate correlation values with dNOTC metric. Apache Lucene, Mina and JBoss Drools exhibit strong correlation values with all test metrics. Therefore, it is normal to observe a moderate-strong correlation between RFC metrics and test metrics in the projects with best source and testing coding.

According to the common trend, RFC metric has strong and moderately strong correlation with both test metrics, therefore it is included in the regression analysis. The inter-metrics correlation table (Table 22) also indicates that RFC metric is strongly correlated to FOUT, LOC_CLS, NOM, TNOM, WMC, LOC_CLS_NEW and NOTC_NEW. This metric has moderate correlation with the rest of the metrics, too.

<u>Specialization Index (SIX) Metric</u>

At the class-level, the number of classes inheriting a specific operation, the number of overridden methods (NORM) and new added methods can also be defined. Related to these measures, the Specialization Index (SIX) metric is defined as:

$$SIX = \frac{NORM * DIT}{NOM}$$

where NOM represents the total number of methods for the class. This measure is useful in differentiating between implementation sub-classing (low values for SIX) and specialization sub-classing (high values of SIX).

NSF metric is one of the other metrics that represent weak correlation values with all test metrics. The new proposed test metrics have different behaviors with respect to their corresponding old metric definitions. When dLOC_CLS metric has, a correlation value of 0.06, dLOC_CLS_NEW metric has a value of 0.09, which means a negligible increase. On the other side, when dNOTC and dLOC_CLS_NEW metrics have both correlation values of 0.08.

For both JBoss and Apache projects, the same pattern applies as stated above. However, it must be noticed that correlation values between SIX metric and dNOTC or JBoss projects only and between SIX metric and dLOCC are insignificant.

Examining the Apache and JBoss projects one by one, we observe that JBoss Drools is the only project that exhibits moderate correlation values with all test metrics, different from the common trend.

According to the common trend, SIX metric is not included in the regression analysis as it exhibits weak results in general. The inter-metrics correlation table (Table 22) also indicates that SIX metric is surprisingly strongly correlated to NORM metric only with a value of 0.98, and has moderate correlation with most of the metrics.

Weighted Methods Per Class (WMC) Metric

WMC metric is one of the metrics that represent moderately strong correlation values with the dLOC_CLS, dLOC_CLS_NEW and dNOTC_NEW test metrics, but moderate values with the dNOTC test metric, for all three cases (single project, Apache and JBoss separately).

The number of methods and the complexity of methods involved is a predictor of how much time and effort is required to develop and maintain the class. The larger the number of methods in a class the greater the potential impact on children, since children will inherit all the methods defined in the class. Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse

The new proposed test metrics have higher correlation values and levels of confidence (meaning lower significance values) with respect to their corresponding old metric definitions. When dLOC_CLS metric has, a correlation value of 0.41, dLOC_CLS_NEW metric has a value of 0.46. Likewise, when dNOTC metric has a correlation value of 0.30, dLOC_CLS_NEW metric has a value of 0.46.

For both JBoss and Apache projects, and considering all projects as a single project case, the new metrics proposed have similar effects with respect to the values of the older metrics with the NOM metric. The difference between dLOC_CLS and dLOC_CLS_NEW metrics is negligible with respect to the change between dNOTC and dNOTC_NEW metrics.

Examining the Apache and JBoss projects one by one, we observe that Apache Ant, Geronimo and JBoss Richfaces exhibit strong correlation values with the dLOC_CLS, dLOC_CLS_NEW and dNOTC_NEW test metrics, and moderate correlation values with dNOTC metric. Apache Lucene, Mina, Wicket and JBoss Drools exhibit strong correlation values with all test metrics. Therefore, it is normal to observe a moderate-strong correlation between WMC metrics and test metrics in the projects with best source and testing coding.

According to the common trend, WMC metric has moderately strong correlation with test metrics, therefore it is included in the regression analysis. This result is expected as one would expect cyclomatic complexity to be related to the testability of the class under consideration. The inter-metrics correlation table (Table 22) also indicates that RFC metric is strongly-moderately correlated to FOUT, LCOM, LOCC, NOM, RFC, TNOM, LOC_CLS_NEW and NOTC_NEW. This metric has moderate correlation with the rest of the metrics, too.

5.4.7.3 Package-Level Metrics

In the following section, we will try to identify the relationship between each source-based class metrics and test suite metrics, so that we can understand how the design parameter the related metric is connected to is affected to testing effort and testability.

In order to obtain a regression analysis between the package-level source-based metrics and test metrics, we will define which metrics will be included in the analysis, according to the correlation and significance values of the correlation analysis we have performed in the previous section. While choosing the metrics to be included in the analysis, we will take into account both in; case level, i.e. single project case, Apache and JBoss cases separately, results and independent project-based results all together.

Below are given the results of correlation analysis we have performed among the package level metrics themselves and between test metrics. Before proceeding, we have to state that, according to the correlation analysis we have performed among all package metrics, test metrics have resulted to have moderate correlation between each other, with a value of 0.49 with a very high level of confidence, 99%.

Table 23 : Correlation Values Among Package Metrics – All as One Single Project

	CA	*CE*	*LOC_PKG*	*NOC*	*NOI*	*RMA*	*RMD*	*RMI*
CA	1.00							
CE	0.33	1.00						
LOC_PKG	0.36	0.69	1.00					
NOC	0.32	0.80	0.76	1.00				
NOI	0.41	0.29	0.23	0.24	1.00			
RMA	0.38	0.17	0.11	0.07	0.79	1.00		
RMD	0.61	0.02	0.11	0.16	0.17	0.16	1.00	
RMI	-0.88	0.04	-0.12	-0.05	-0.31	-0.34	-0.69	1.00

Table 24 : Correlation & Significance Results of All Projects as One Project – Package Level

Correlation Coefficients ρ(x; y)	**dLOC-PKG**	**dNOTC-PKG**
CA	0.276231	0.037849
CE	0.485595	0.238211
NOC	0.530785	0.231695
NOI	0.211318	0.030912
RMA	0.069718	-0.0375
RMD	0.124646	-0.06925
RMI	-0.15653	0.070915
LOC_PKG	0.493095	0.180649

Significance Values p(x; y)	**dLOC-PKG**	**dNOTC-PKG**
CA	4.42E-21	0.004482
CE	2.20E-67	1.47E-73
NOC	1.75E-82	1.36E-69
NOI	8.79E-13	0.020282
RMA	0.01957	0.004862
RMD	2.85E-05	1.94E-07
RMI	1.38E-07	9.83E-08
LOC_PKG	9.81E-70	1.48E-42

Table 25 : Correlation & Significance Results of JBoss Projects Only – Package Level

Correlation Coefficients ρ(x; y)	**dLOC-PKG**	**dNOTC-PKG**
CA	0.214047	-0.12103
CE	0.419793	0.130044
NOC	0.469181	0.113017
NOI	0.125852	-0.11612
RMA	-0.00468	-0.13834
RMD	0.084555	-0.19065
RMI	-0.10642	0.182474
LOC_PKG	0.42526	0.030749

Significance Values p(x; y)	**dLOC-PKG**	**dNOTC-PKG**
CA	2.11E-06	8.47E-12
CE	5.34E-22	2.08E-13
NOC	9.37E-28	1.83E-10
NOI	0.00566	5.71E-11
RMA	0.918356	5.44E-15
RMD	0.063615	2.80E-27
RMI	0.019437	4.32E-25
LOC_PKG	1.37E-22	0.083747

Table 26 : Correlation & Significance Results of Apache Projects Only – Package Level

Correlation Coefficients ρ(x; y)	**dLOC-PKG**	**dNOTC-PKG**
CA	0.32311	0.197062
CE	0.536904	0.345647
NOC	0.588147	0.349822
NOI	0.271647	0.18068
RMA	0.128698	0.070762
RMD	0.149275	0.069376
RMI	-0.19642	-0.04797
LOC_PKG	0.554994	0.332867

Significance Values p(x; y)	**dLOC-PKG**	**dNOTC-PKG**
CA	5.41E-17	4.51E-23
CE	5.39E-49	2.49E-70
NOC	1.02E-60	3.89E-72
NOI	2.85E-12	1.34E-19
RMA	0.001112	0.000429
RMD	0.000152	0.000556
RMI	5.6E-07	0.017045
LOC_PKG	6.67E-53	4.68E-65

Richtmyer-Meshkov Abstractness (RMA) Metric

This metrics defines the number of abstract classes (and interfaces) divided by the total number of types in a package. It is also one of the metrics that exhibit the weakest correlation values with insignificant levels of confidence.

Apache projects only case have more significant and better correlation values with respect to the case of considering all projects as a single project case. JBoss projects only case have insignificant values for dLOC_PKG metric, whereas single project and Apache only cases have weak correlation values with this metric. For the dNOTC_PKG test metric, the single project and JBoss projects only cases have weak negative correlation values, whereas the Apache projects only case have positive and greater but still weak correlation values. The results of the three cases represent either weak or insignificant correlation values, which mean that this metric is not correlated to the test metrics, and should be discarded in the regression analysis. Examining the Apache and JBoss projects one by one does not change our opinion, as many results are insignificant and contradictory with other projects.

This result is somehow surprising as one would expect abstractness to be related to the testability of the package under consideration. Under normal circumstances, according to how prone the package is to modification during the application's life cycle, it must be abstract to a greater or lesser extent. The more stable a package must be, the more abstract it must be, if it is to be extensible. Abstract packages that are extensible provide greater model flexibility. Thus, the fact that testability and abstraction are uncorrelated makes as important insight to our way of thinking.

According to the common trend, RMA metric is not included in the regression analysis as it has insignificant and very weak correlation with both test metrics. This result seems to be surprising as one would expect abstractness of a package to be related to the testability of the package under consideration. The inter-metrics correlation table (Table 23) also indicates that RMA metric is strongly correlated to NOI metric only. This metric has moderate correlation with CA and RMA metrics, and weak correlation with rest of the metrics.

Afferent Coupling (CA) Metric

This metric defines the number of classes outside a package that depend on classes inside the package. It measures the number of types outside a package that depend on types within the package (incoming dependencies). High afferent coupling indicates that the concerned packages have many responsibilities.

Afferent coupling allows one to more effectively evaluate the cost of change and the likelihood of reuse. For instance, maintaining a module with many incoming dependencies is more costly and risky since there is greater risk of impacting other modules, requiring more effort thorough integration testing. Conversely, a module with many outgoing dependencies is more difficult to test and reuse since all dependent modules are required.

Concrete modules with high afferent coupling will be difficult to change because of the high number of incoming dependencies. Modules with many abstractions are typically more extensible, so long as the dependencies are on the abstract portion of a module.

The case of considering all projects as a single project case indicates different results in terms of correlation values. For this case, the CA metric has weak and negligible correlation with the dNOTC_PKG test metric, whereas the Apache and JBoss projects only cases and analysis of all projects standalone indicate that there exists moderate correlation with high levels of confidence. Therefore, it seems better to ignore the single-project case and focus on the other two cases and standalone project examinations.

Apache projects show better correlations between CA and the two test metrics with respect to JBoss projects, i.e. 0.32 versus 0.21 for dLOC_PKG metric and 0.19 versus 0.12 for dNOTC_PKG metric respectively. The correlation results also showed that, for most of the projects and all three cases, CA is a significantly better predictor of the dLOC_PKG metric than of the dNOTC_PKG metric. Thus, the association between the afferent coupling of a package and the size of its test suite is significantly stronger than the association between the afferent coupling and the number of test cases.

Examining the Apache and JBoss projects one by one, we observe that Apache Ant, Mina, Wicket and JBoss Cache exhibit strongly moderate correlation values with dLOC_PKG test metric, and all three cases (single project, Apache and JBoss separately), Apache Lucene, Geronimo, JBoss Drools and Richfaces exhibit moderate correlation values with the same metric. For the other test metric, dNOTC_PKG, we observe that Apache Only case, Apache Ant, Lucene, Geronimo, Wicket and JBoss Drools exhibit moderate correlation values with the dLOC_PKG test metric, whereas Apache Mina is the only project that exhibits strongly moderate correlation values with the same metric.

According to the common trend, CA metric has moderate correlation with both test metrics, therefore it is included in the regression analysis. This result is expected as one would expect coupling to be related to the testability of the package under consideration.

The inter-metrics correlation table (Table 23) also indicates that CA metric is strongly correlated to RMD and RMI metrics. This metric has moderate correlation with most of the rest. Surprisingly, it has very strong correlation with two different metrics other than CE metric, which indicates afferent and efferent coupling are not correlated strongly, as one may expect.

Efferent Coupling (CE) Metric

This metrics defines the number of classes inside a package that depend on classes outside the package. It measures the number of types inside a package that depends on types outside of the package (outgoing dependencies). High efferent coupling indicates that the concerned package is dependant.

Efferent coupling allows one to more effectively evaluate the cost of change and the likelihood of reuse. For instance, maintaining a module with many incoming dependencies is more costly and risky since there is greater risk of impacting other modules, requiring more effort thorough integration testing. Conversely, a module with many outgoing dependencies is more difficult to test and reuse since all dependent modules are required.

Concrete modules with high efferent coupling will be difficult to change because of the high number of incoming dependencies. Modules with many abstractions are typically more extensible, so long as the dependencies are on the abstract portion of a module.

All three cases (single project, Apache and JBoss separately) exhibit strongly moderate correlation values with dLOC_PKG test metric and moderate correlation values with dNOTC_PKG test metric. All correlation values are significant with high levels of confidence.

Apache projects show better correlations between CE and the two test metrics with respect to JBoss projects, i.e. 0.53 versus 0.42 for dLOC_PKG metric and 0.34 versus 0.13

for dNOTC_PKG metric respectively. The correlation results also showed that, for most of the projects and all three cases, CE is a significantly better predictor of the dLOC_PKG metric than of the dNOTC_PKG metric. Thus, the association between the efferent coupling of a package and the size of its test suite is significantly stronger than the association between the efferent coupling and the number of test cases.

Examining the Apache and JBoss projects one by one, we observe that all Apache projects and JBoss Cache exhibit strong-strongly moderate correlation values with dLOC_PKG test metric, and all three cases (single project, Apache and JBoss separately), JBoss Drools and Richfaces exhibit moderate correlation values with the same metric. For the other test metric, dNOTC_PKG, we observe that all Apache projects except for Wicket, JBoss Drools and Cache exhibit strong-strongly moderate correlation values with the dLOC_PKG test metric, whereas that all three cases indicate moderate correlation with the same metric.

According to the common trend, CE metric has strong-moderately strong correlation with dLOC_PKG test metric and has moderate correlation with dNOTC_PKG test metric, therefore it is included in the regression analysis. This result is expected as one would expect coupling to be related to the testability of the package under consideration.

The inter-metrics correlation table (Table 23) also indicates that CE metric is strongly correlated to LOC_PKG and NOC metrics. This metric has moderate correlation with CA metric. Surprisingly, it has very strong correlation with two different metrics other than CA metric, which indicates afferent and efferent coupling are not correlated strongly, as one may expect.

Richtmyer-Meshkov Instability (RMI) Metric

This metric is an indicator of the package's resilience to change. The range for this metric is 0 to 1, with RMI = 0 indicating a completely stable package and RMI = 1 indicating a completely instable package.

A package is that much more unstable if it depends more on other packages than they depend on it. It is likely to change if these other packages change. Each value calculated for a given package must be compared to the values of the other packages. Not all packages have to be stable, since it must be possible for the application to evolve. If the user wishes the package to be stable, it must depend less on the other packages than they depend on it.

All three cases (single project, Apache and JBoss separately) exhibit strongly moderate correlation values with dLOC_PKG test metric and moderate correlation values with dNOTC test metric. All correlation values are significant with high levels of confidence.

Apache projects show better correlations between RMI and the dLOC_PKG test metric with respect to JBoss projects, i.e. -0.19 versus -0.10 respectively, JBoss projects show better correlations between RMI and the dNOTC_PKG test metric with respect to Apache projects, i.e. 0.18 versus -0.04 respectively. The correlation results also showed that, for most of the projects and all three cases, RMI is a significantly better predictor of the dLOC_PKG metric than of the dNOTC_PKG metric. Thus, the association between the package's resilience to change and the size of its test suite is significantly stronger than the association between resilience to change and the number of test cases.

Examining the Apache and JBoss projects one by one, we observe that all three cases (single project, Apache and JBoss separately) and all projects except for Apache Wicket exhibit weak-moderate correlation values with dLOC_PKG test metric, Apache Wicket is the only project that exhibits strong correlation value with the same metric. For the other test metric, dNOTC_PKG, we observe that all single project case, Apache-only case and most of the projects exhibit weak correlation values with the dNOTC_PKG test metric, whereas JBoss-only case, Apache Mina, Wicket exhibit moderate correlation with the same metric.

According to the common trend, RMI metric has moderately weak correlation with both test metrics, nevertheless we partially, i.e. for some test metrics only, include this metric in the regression analysis, as the independent project results indicate some good correlation. The correlation results are somehow surprising, as one would expect stability to be related to the testability of the package under consideration.

The inter-metrics correlation table (Table 23) also indicates that RMI metric is strongly correlated to CA and RMD metrics. This metric has moderate correlation with RMA and NOI metrics, and weak correlation with rest of the metrics.

<u>Normalized Distance from Main Sequence (RMD) Metric</u>

This metric measures the balance between the abstraction and instability rates of the package, i.e. how far away a category is from this ideal. According to what function a package has to perform, it must be able to be unstable, in other words, often significantly or abstractly modified. It must be sufficiently general to be adaptable to widely diverse situations, either without being modified or with only minimal modifications. It is preferable to have a balance between these contradictory criteria.

All three cases (single project, Apache and JBoss separately) exhibit weak correlation values with both test metrics. Most correlation values are significant with high levels of confidence.

Single project case and Apache projects only resemble to each other for dLOC_PKG metric but diverges in dNOTC_PKG metric. For dNOTC_PKG metric, they have the same correlation values but with inverse signs. On the other hand, JBoss projects only case have less correlation value with dLOC_PKG metric with respect to the other two cases, but has better correlation value with dNOTC_PKG metric under the same direction of relation, i.e. same sign, both negative.

Examining the Apache and JBoss projects one by one, we observe Apache Ant and Wicket exhibit moderate correlation values with both test metrics, and Apache Mina exhibit moderate correlation value with the dNOTC_PKG metric for a low level of confidence, 90%. The results of the three cases and standalone project analysis represent either weak or insignificant correlation values, which mean that this metric is not correlated to the test metrics, and should be discarded in the regression analysis.

According to the common trend, RMD metric is not included in the regression analysis as it has insignificant and very weak correlation with both test metrics. This result seems to be adequate as this metric measures the balance between the abstraction and instability rates of the package, and both abstractness and instability have been found to be uncorrelated to package testability using their indicatory metrics.

The inter-metrics correlation table (Table 23) also indicates that RMD metric is strongly correlated to CA and RMI metrics. This metric has weak correlation with rest of the metrics.

Number of Classes (NOC) Metric

This metrics defines the total number of classes inside a package. High values mean high memory footprint, higher complexity but high modularity too. Lower values can lead to poor application design but better system physical proprieties.

All three cases (single project, Apache and JBoss separately) exhibit strongly moderate correlation values with dLOC_PKG test metric and moderate correlation values with dNOTC_PKG test metric. All correlation values are significant with high levels of confidence. Apache projects show better correlations between NOC and both test metrics with respect to JBoss projects, i.e. 0.58 versus 0.46 for dLOC_PKG metric and 0.35 versus 0.11 for dNOTC_PKG metric respectively. The correlation results also showed that NOC is a significantly better predictor of the dLOC_PKG metric than of the dNOTC_PKG metric. Thus, the association between the number of classes in a package and the size of its test suite is significantly stronger than the association between the number of classes and the number of test cases.

Examining the Apache and JBoss projects one by one, we observe that all Apache projects, JBoss Cache and Richfaces exhibit strong and moderately strong correlation values with dLOC_PKG test metric, and JBoss Drools is the only project that exhibits moderate correlation value with the same metric. For the other test metric, dNOTC_PKG, we observe that Apache Ant, Lucene, Mina, JBoss Cache and Drools exhibit strong and moderately strong correlation values with the dNOTC_PKG test metric, whereas Apache Geronimo and Wicket exhibit moderate correlation with the same metric.

According to the common trend, NOC metric is included in the regression analysis as it has significant and promising correlation with both test metrics. This result is normal, as one would expect size of a package in number of classes to be directly related to the testability of the package under consideration.

The inter-metrics correlation table (Table 23) also indicates that NOC metric is strongly correlated to CE and LOC_PKG metrics. This metric has moderate correlation with CA and NOI metrics, and weak correlation with rest of the metrics.

Number of Interfaces (NOI) Metric

This metrics defines the total number of interfaces inside a package. Higher number of methods means more modularization (assuming two solutions with the same Method of Lines of Code) and this lead to a more readable solution but also mean more method calls. (that can greatly reduce performance)

The case of considering all projects as a single project case indicates different results in terms of correlation values. For this case, the RMI metric has weak and negligible correlation with the dNOTC_PKG test metric, whereas the Apache and JBoss projects only cases indicate that there exists weak-moderate correlation with high levels of confidence but with inverse signs. Apache projects show better correlations between NOI and both test metrics with respect to JBoss projects, i.e. 0.27 versus 0.12 for dLOC_PKG metric and 0.18 versus -0.11 for dNOTC_PKG metric respectively. The correlation results also showed that NOI is a significantly better predictor of the dLOC_PKG metric than of the dNOTC_PKG metric. Thus, the association between the number of interfaces in a package and the size of its test suite is significantly stronger than the association between the number of interfaces and the number of test cases.

Examining the Apache and JBoss projects one by one, we observe that Apache Ant, Wicket, JBoss Cache exhibit moderately strong correlation values with dLOC_PKG test metric, and, Apache Geronimo, JBoss Drools and Richfaces exhibit moderate correlation value with the same metric. For the other test metric, dNOTC_PKG, we observe that Apache

Ant, Lucene, Wicket, JBoss Cache and Drools exhibit moderate correlation values with the dNOTC_PKG test metric, whereas the rest exhibit insignificant and negligible correlation with the same metric.

According to the common trend, NOI metric is included in the regression analysis as it has significant and promising correlation with both test metrics. This result is normal as one would expect size of a package in number of interfaces to be directly related to the testability of the package under consideration.

The inter-metrics correlation table (Table 23) also indicates that NOI metric is strongly correlated to RMA metric only. This metric has moderate correlation with most of the metrics left.

Lines Of Code per Package (LOC_PKG) Metric

Different from the class-level correlation results, LOC_PKG metric is not the metric that represent the highest correlation values with the test metrics

A common basis of estimate on a software project is lines of code metrics. Lines of code are used to create time and cost estimates. The class-level and package-level estimates become the baseline to measure the degree of work performed on a project. Once a project is underway, the LOC metrics becomes a tracking tool to measure the degree of progress on a module or project. The LOC measurement becomes the barometer for the program's progress and productivity.

For the case of considering all projects as a single project, LOC_PKG metric represents moderately strong correlation values with dLOC_PKG test metric. However, for all three cases (single project, Apache and JBoss separately), the correlation values differ for the dNOTC_PKG. On the other side, single project case and Apache projects only case exhibit moderate correlation, whereas JBoss projects exhibit very weak correlation values with a low level of confidence. Analyzing all of the projects one by one, the problem seems to be JBoss Richfaces project among the JBoss projects as it has insignificant correlation measurement. It seems better to continue after discarding this project in JBoss only project consideration. The new correlation shows a similar pattern to Apache only project and the single project case values approach to the Apache projects only case.

Examining the Apache and JBoss projects one by one, we observe that all projects (discarding JBoss Richfaces for dNOTC_PKG metric evaluation) exhibit moderately strong and strong correlation values with both test metrics, as one would normally expect.

According to the common trend, LOC_PKG metric is included in the regression analysis as it has significant and promising correlation with both test metrics. This result is normal, as one would expect size of a package to be directly related to the testability of the package under consideration.

The inter-metrics correlation table (Table 23) also indicates that LOC_PKG metric is strongly correlated to CE and NOC metrics. This metric has moderate correlation with CA and NOI metrics, and weak correlation with rest of the metrics.

5.4.8 Project-Level Metrics

We have examined the two project-level metrics with the two test metrics, just similar to the class and package level analysis we have performed before. The correlation analysis indicates that the total lines of code in the project is very strongly related to the required lines of test codes in the project, as one would normally expect. The results show that the total lines of code in the project has a strongly moderate correlation with the number of test cases to exist in the project test suites and cases.

The number of packages that exist in a project has a strongly moderate correlation with the required lines of test codes in the project test suites and cases, similar to TLOC and dNOTC relationship. This metric exhibits a moderate correlation with the number of test cases to exist in the project test suites and cases.

All three cases (single project, Apache and JBoss separately) exhibit similar results, meaning the results are independent of the context the software is developed and we can make a generalization with these results. The Apache Projects Only case insignificant correlation value between NOP and both test metrics. Therefore, eliminating these results, we may restate that the relation is stronger than we have stated above.

The correlation results also showed that TLOC is a significantly better predictor of the dLOCC metric than of the dNOTC metric. Thus, the association between the line count of a project and the size of its test suite is significantly stronger than the association between the line count and the number of test cases.

Similarly, the correlation results also showed that NOP is a significantly better predictor of the dLOCC metric than of the dNOTC metric. Thus, the association between the number of packages in a project and the size of its test suite is significantly stronger than the association between the number of packages and the number of test cases.

Table 27 : Correlation & Significance Values Among Project Metrics – All As One Single Project

Correlation Values				Significance Values		
	dLOC	**dNOTC**			**dLOC**	**dNOTC**
TLOC	0.966	0.586		***TLOC***	1.30E-24	5.60E-05
NOP	0.598	0.349		***NOP***	3.54E-05	0.0252

Table 28 : Correlation & Significance Values Among Project Metrics – Apache Projects Only

Correlation Values				Significance Values		
	dLOC	**dNOTC**			**dLOC**	**dNOTC**
TLOC	0.891	0.575		***TLOC***	3.62E-06	1.97E-02
NOP	0.347	0.263		***NOP***	0.1878	0.3242

Table 29 : Correlation & Significance Values Among Project Metrics – JBoss Projects Only

Correlation Values				Significance Values		
	dLOC	**dNOTC**			**dLOC**	**dNOTC**
TLOC	0.980	0.618		***TLOC***	8.01E-18	9.90E-04
NOP	0.715	0.489		***NOP***	5.75E-05	0.0129

5.4.9 Regression Analysis

Using the correlation coefficients and significance values among the source metrics in two different classification levels, i.e. class and package levels, we have defined the metrics that are correlated to each other, preferring stronger correlations over weaker ones and more significant results to non-significant ones.

We have seen from the statistical results that there exist relationships among the source-based metrics and test suite metrics. Due to mathematical simplicity, we have assumed that a multiple linear equation exists among these metric sets. Then, a multiple linear regression has been used to get the coefficients of this equation. The multiple linear regressions establish a relationship between dependent variables and multiple independent variables. The regression equation takes the form:

$$y = \beta_0 + \beta_1 x_1 + \ldots\ldots + \beta_m x_m$$

where "x"s represent the independent variables, i.e. our current source-based class-level or package-level metrics, "y" is the dependent variable, i.e. our test metric and "β"s represent the regression coefficients, which indicate the net effect the independent variable on the dependent variable, holding the remaining variables in the equation constant. Component-wise effect may be speculated and respective component weightings (CW) may be fixed using regression equation. Thereby, the CWs of individual design parameters have been calculated in terms of regression coefficient β.

The statistical assessment of the correlation analysis results have shown that, the different sets of metrics are adequate for different test metrics to be taken into consideration in the regression analysis to be performed in both class and package levels. These metrics are indicated as bold in the corresponding regression analysis tables below. The expected test metric value is calculated by summing products of regression coefficients and the class or package metrics in the set of the corresponding test metric.

$$\textbf{\textit{Expected Test Metric Value}} = \sum(Coefficient * Metric\ Value)$$

5.4.9.1 Package-Level Analysis

The regression analysis we have performed in package level have produced the following equations for obtaining the expected metric values so that we can conclude that the packages are adequate to be tested properly and necessarily.

<u>LOC_PCKAGE_TEST</u> =

```
- 0.8528 * CA + 95.0431 * NOI - 1627.9391* RMA
+ 0.2802* LOC_PACKAGE
```

<u>NOTC_PACKAGE_TEST</u> =

```
- 1.9674 * CE + 11.9214 * NOI - 207.6078 * RMA
+ 32.5975 * RMD + 23.1810 *  RMI
+ 0.0569 * LOC_PACKAGE
```

The following regression details show the results of our regression analysis for all three cases, i.e. single project, Apache and JBoss separately. We have submitted and evaluated all three cases in order to see the effects of the context the software is developed into the metric requirements. We have calculated the expected metric values with the correlation coefficients and results by producing different equations for three cases with the corresponding regression coefficients.

In order to how much the expected values differ with the corresponding equations peculiar to the three cases, we have performed correlation analysis among the expected test metrics values. Table 27, Table 28 and Table 29 show the correlations among the test metric sets of the three cases separately.

Similar to the class-level results, the correlation values of the case considering all projects as one single project is more correlated to the Apache projects only case with respect to the JBoss projects only case.

Table 30 : Regression Analysis for Test Metrics Values - All Projects as One Single Project Case – Package Level

dLOC-PKG				**dNOTC-PKG**		
	Coefficients	***P-value***			***Coefficients***	***P-value***
CA	**-0.8528**	0.0607		**CA**	-0.0741	0.3908
CE	-4.7186	0.2558		**CE**	**-1.9674**	0.0130
NOC	1.2375	0.5904		**NOC**	-0.2261	0.6053
NOI	**95.0431**	4.97756E-14		**NOI**	**11.9214**	4.27384E-07
RMA	**-1627.9391**	8.09464E-07		**RMA**	**-207.6078**	0.0009
RMD	102.6484	0.2999		**RMD**	**32.5975**	0.0839
RMI	55.1153	0.3388		**RMI**	**23.1810**	0.0348
LOC_PKG	**0.2802**	1.08002E-20		**LOC_PKG**	**0.0569**	4.34296E-23

5.4.9.2 Class-Level Analysis

The regression analysis we have performed in class level have produced the following equations for obtaining the expected metric values and concluding that the source class-test class pair has a healthy relationship to be tested properly and necessarily.

LOC_CLASS_TEST =

$$6.6672 * \mathbf{DIT} + 4.3128 * \mathbf{FOUT} + 5.0025 * \mathbf{NORM} + 2.2659 * \mathbf{NSF} - 0.9831 * \mathbf{RFC} - 16.7822 * \mathbf{SIX} - 1.9227 * \mathbf{TNOF} + 0.7303 * \mathbf{WMC}$$

LOC_CLASS_TEST_NEW =

```
4.6033 * DIT + 3.0515 * FOUT + 9.4379 * LCOM
+ 1.3550 * NSF – 0.6470 * RFC – 8.8597 * SIX
– 1.0034 * TNOF + 0.5780 * WMC
```

NOTC_CLASS_TEST =

```
0.7822 * DIT + 0.6295 * FOUT + 1.6239 * NORM
+ 0.8967 * NSF – 0.1630 * RFC – 4.6250 * SIX
– 1.0067 * TNOF + 0.4202 * TNOM + 0.0967 * WMC
```

NOTC_CLASS_TEST_NEW =

```
3.2002 * DIT + 2.1229* FOUT + 6.4686 * LCOM
+ 0.9472 * NSF – 0.4444 * RFC – 6.2000 * SIX
– 0.7051 * TNOF + 0.3992 * WMC
```

The following regression details show the results of our regression analysis for all three cases, i.e. single project, Apache and JBoss separately. We have submitted and evaluated all three cases in order to see the effects of the context the software is developed into the metric requirements. We have calculated the expected metric values with the correlation coefficients and results by producing different equations for three cases with the corresponding regression coefficients.

In order to how much the expected values differ with the corresponding equations peculiar to the three cases, we have performed correlation analysis among the expected test metrics values.Table 45, Table 46 and Table 47 show the correlations among the test metric sets of the three cases separately.

The JBoss projects only case includes three significant projects, whereas Apache projects only case includes five significant projects. The total lines of code for the Apache projects is nearly 1.5 bigger than the JBoss case. Also, the testing effort and coding is better in Apache projects with respec to the JBoss case. The correlation values support this fact, as the case considering all projects as one single project is more correlated to the Apache projects only case with respect to the JBoss projects only case.

Table 45 : Correlation Between Expected Regression Test Values : All Projects as One Single Project Case versus JBoss Projects Only Case – Package Level

CORRELATION VALUES		***SINGLE PROJECT***	
		dLOC_PKG	**dNOTC_PKG**
JBOSS ONLY	**dLOC_PKG**	**0.746**	0.695
	dNOTC_PKG	0.421	**0.469**

Table 46 : Correlation Between Expected Regression Test Values : All Projects as One Single Project Case versus Apache Projects Only Case – Package Level

CORRELATION VALUES		**SINGLE PROJECT**	
		dLOC**_PKG**	**dNOTC_PKG**
APACHE ONLY	**dLOC_PKG**	0.986	**0.988**
	dNOTC_PKG	0.975	**0.996**

Table 47 : Correlation Between Expected Regression Test Values : JBoss Projects Only Case versus Apache Projects Only Case – Package Level

CORRELATION VALUES		**APACHE ONLY**	
		dLOC**_PKG**	**dNOTC_PKG**
JBOSS ONLY	**dLOC_PKG**	0.635	**0.611**
	dNOTC_PKG	0.361	**0.355**

Table 31 : Regression Analysis for Test Metrics Values - All Projects as One Single Project Case – Class Level

dLOC_CLS			**dLOC_CLS_NEW**			**dNOTC**			**dNOTC_NEW**		
	Coefficients	***P-value***		***Coefficients***	***P-value***		***Coefficients***	***P-value***		***Coefficients***	***P-value***
DIT	**6.6672**	2.32E-06	**DIT**	**4.6033**	7.09E-08	**DIT**	**0.7822**	0.0105	**DIT**	**3.2002**	7.21E-08
FOUT	**4.3128**	2.13E-08	**FOUT**	**3.0515**	6.08E-11	**FOUT**	**0.6295**	0.0002	**FOUT**	**2.1229**	6.03E-11
LCOM	8.5444	0.3392	**LCOM**	**9.4379**	0.0808	**LCOM**	0.0899	0.9631	**LCOM**	**6.4686**	0.085
LOCC	0.0351	0.5761	**LOCC**	-0.0073	0.8473	**LOCC**	0.0221	0.1049	**LOCC**	-0.0053	0.841
NOF	0.0000	0	**NOF**	0.0000	0	**NOF**	0.0000	0	**NOF**	0.0000	0
NOM	0.9834	0.2187	**NOM**	0.3818	0.4293	**NOM**	-0.0508	0.7698	**NOM**	0.2587	0.441
NORM	**5.0025**	0.0494	**NORM**	2.5177	0.1016	**NORM**	**1.6239**	0.0033	**NORM**	1.7429	0.103
NSC	0.5854	0.3416	**NSC**	0.4594	0.2170	**NSC**	-0.1032	0.4401	**NSC**	0.3119	0.228
NSF	**2.2659**	0.0215	**NSF**	**1.3550**	0.0229	**NSF**	**0.8967**	2.94E-05	**NSF**	**0.9472**	0.022
NSM	0.0000	0	**NSM**	0.0000	0	**NSM**	0.0000	0	**NSM**	0.0000	0
RFC	**-0.9831**	0.0003	**RFC**	**-0.6470**	7.34E-05	**RFC**	**-0.1630**	0.0054	**RFC**	**-0.4444**	9.01E-05
SIX	**-16.7822**	0.0074	**SIX**	**-8.8597**	0.0192	**SIX**	**-4.6250**	0.0007	**SIX**	**-6.2000**	0.018
TNOF	**-1.9227**	0.0235	**TNOF**	**-1.0034**	0.0504	**TNOF**	**-1.0067**	5.55E-08	**TNOF**	**-0.7051**	0.048
TNOM	0.2383	0.7835	**TNOM**	0.3025	0.5637	**TNOM**	**0.4202**	0.0258	**TNOM**	0.2213	0.544
WMC	**0.7303**	0.0045	**WMC**	**0.5780**	0.0002	**WMC**	**0.0967**	0.0830	**WMC**	**0.3992**	0.000

CHAPTER 6

VALIDATION OF THE MODEL

This chapter provides the details on how we have validated our model. It presents the results and assessments of the validation process, as well.

6.1 About Validation Process

We have calculated the expected values for the test metrics and compared these values with the observed values in the source code. In order to make a comparison, we have tried to examine how much the observed values diverge from the expected values. Below is shown how we have calculated this ratio, the divergence ratio.

$$\boldsymbol{Divergence\ Ratio} = \frac{(\boldsymbol{Expected\ Value} - \boldsymbol{Observed\ Value})}{(\boldsymbol{Expected\ Value})}$$

As seen from the formula, ratios close to zero or with negative sign mean that the expected and observed values are very close to each other, or the observed testing effort is even more than expected by our model. The divergence ratio of magnitude one means that the associated class-test class pair does not exist, i.e. the class is not tested in a separate test class, or the package has no testing code inside.

As our model is a linear regression, it may produce faulty results for some cases, such as negative expected values for class or package level metric values. These faulty results cause divergence ratios greater than one. These faulty results can be easily seen on the scatter charts of deviations to be shown below. They exist only in the class-level measurements, and are only a few, so they may be neglected. Our model may be stated to be strong as it produces insignificant faulty results.

6.2 Package-Level Analysis

The correlation values calculated among the test metrics expected values for the three cases indicated that, the case considering all projects as one single project is significant and promising enough to make a generalization such that the expected test metric value equations can be used for any project, independent of the context. To support our opinion, we have applied the equations for all eight significant projects and observed that it really points out the packages that have not been tested as necessary.

The expected values and the real values differ most of the time, but by defining a maximum value to the divergence ratio for the results to diverge from the expected value, we have accepted that we allow the values to differ with a level of a safety margin. For example, we may set a maximum divergence ratio of "0.5". This means, we accept observed values more than or equal to half of the expected values. The two test metrics may have different maximum values for allowed divergence ratios, as they scatter different from each other.

We have also used 7 new projects more that have not been used before in our study to see whether our expected value equations successfully identifies non-conforming packages. These projects either had insignificant number of class-test class pairs or did not have proper source code releases including testing capabilities at the times we have determined the projects to use in our study.

Our observation is that the model helps to identify successfully the non-conforming pairs. Nevertheless, the model allows the user to decide whether the pair is non-conforming as the safety margin may vary from person to person and from organization to organization.

The correlation value between the expected number of test cases and size of test suite was found as 0.99, which means that the expected values our equations require for the test suite completely positively correlated.

The following graphs show the scatter chart of the difference from the expected values for all three cases, i.e. single project, Apache and JBoss projects only separately. The APPENDIX A includes the rest of the graphs for all projects we have used to validate our model. The equations have resulted very high negative values for some packages, and they have been omitted on the charts to have a better look.

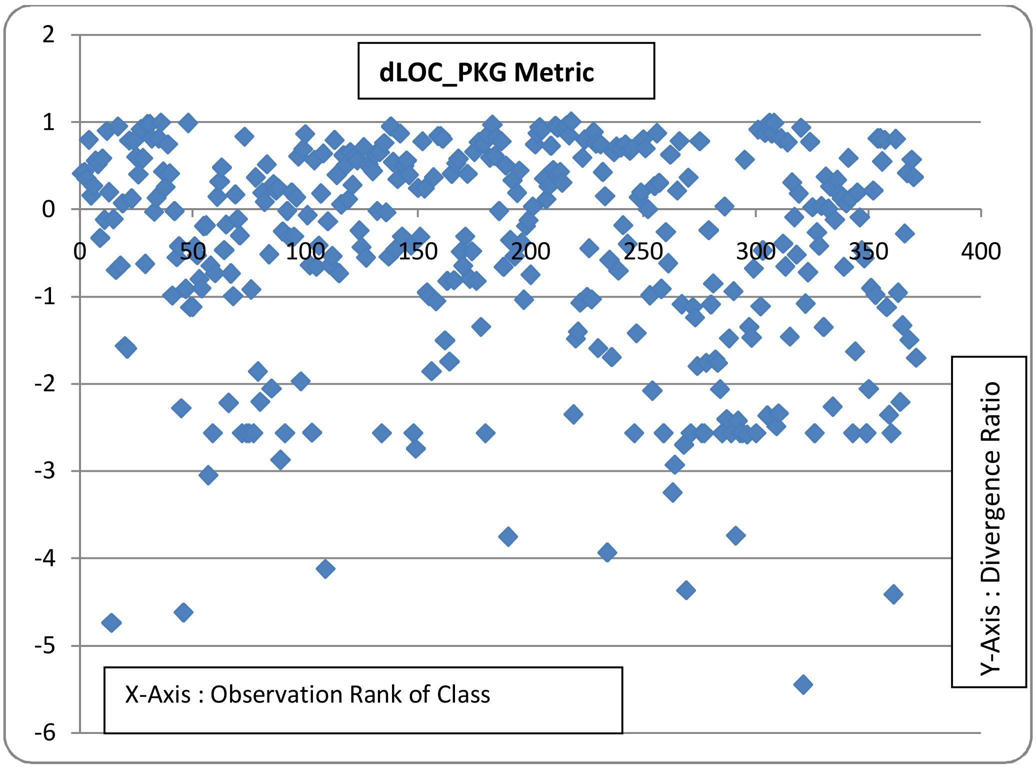
Examining the three cases (single project, Apache and JBoss projects only) separately, we observe that most of the measurements for the expected test metric values is less than or equal to zero, meaning the testing effort for these packages is satisfactory. It is easily seen from the chart that the number of packages having no test cases are more than the number of packages having no test codes. This implies that these test codes do not contain test cases, but test the related classes with other test statements other than assertments.

For the Apache Ant project, most of the packages seem to contain more than half of the necessary amount of test lines of code. Whereas, the number of test cases chart imply that most of the packages have less than half of the necessary amount of number of test cases. We may state that the testing effort must be increased, especially with more test case assertment statements to have a better testing structure of the packages. For the Apache Lucene project, the majority of the packages have a divergence ratio less than or equal to zero, meaning test lines of code is adequate. The NOTC chart implies the same consequences as the Ant project, as stated above.

The Apache Mina and Wicket, JBoss Drools and Richfaces projects indicate similar patterns for both metrics like the Lucene project. JBoss Cache project seems to be the most successful project among the eight old projects in both test metric divergence ratio charts. Nearly all the packages have a divergence ratio less than or equal to zero for dLOC_PKG metric, and most of the packages have a ratio close to or less than zero for dNOTC_PKG metric. Whereas, Apache Geronimo project seems to be the worst of the 8 projects as half of the ratios for dLOC_PKG metric and most the ratios for dNOTC_PKG metric are between 0 and 1.

Among the new seven projects used to test our model, Apache ActiveMQ, OJB, Struts and JBoss ESB indicate successful results for dLOC_PKG metrics, as most of the ratios are either negative or very close to zero. For the dNOTC_PKG metric, Apache OJB shows the most successful results, and most of the projects left indicate a similar pattern to the case of the eight projects, implying that number of test cases should be increased for most of the packages.

The scatter charts of deviation used in our study aims to visualize and simplify the scattering of our deviation observation for both package and class level measurements. The vertical axis measures the value of divergence ratio, which has been explained above, and the horizontal axis represents the rank of the related observation point. The chart below displays the divergence ratios of nearly 375 observation points, as seen from the maximum observation point. The observation points are displayed separately, to indicate the frequencies of the observed ratios. In case only observed ratios would be displayed on the vertical axis only, this valuable data would be lost.



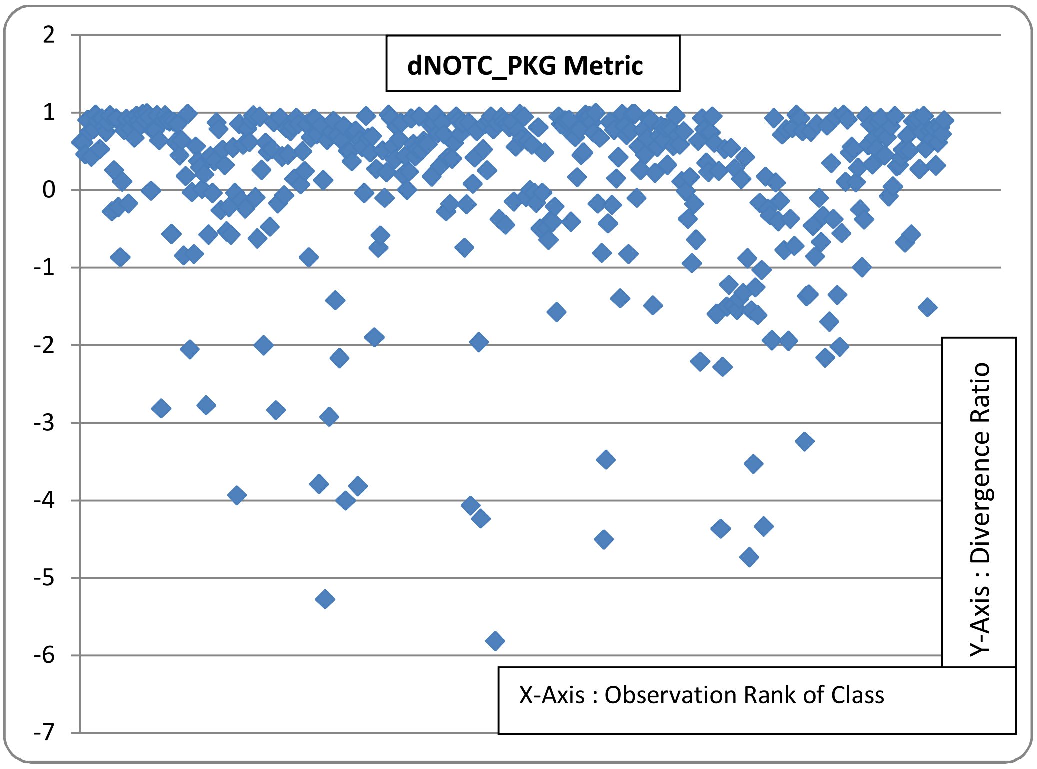


Figure 4 : Scatter Charts of Deviation - Test Metrics - Package Level - All Projects as One Single Project

6.3 Class-Level Analysis

The correlation values calculated among the test metrics expected values for the three cases indicated that, the case considering all projects as one single project is significant and promising enough to make a generalization such that the expected test metric value equations can be used for any project, independent of the context. To support our opinion, we have applied the equations for all eight significant projects and observed that it really points out non-conforming couples.

We have also used 9 new projects more that have not been used before in our study to see whether our expected value equations successfully identifies the source class-test class pairs. These projects either had insignificant number of class-test class pairs or did not have proper source code releases including testing capabilities at the times we have determined the projects to use in our study.

Similar to the package-level analysis, the expected values and the real values differ most of the time, but by defining a maximum value to the divergence ratio for the results to diverge from the expected value, we have accepted that we allow the values to differ with a level of a safety margin. For example, we may set a maximum divergence ratio of “0.5”. This means, we accept observed values more than or equal to half of the expected values. The two test metrics may have different maximum values for allowed divergence ratios, as they scatter different from each other.

Our observation is that the model helps to identify successfully the non-conforming pairs. Nevertheless, the model allows the user to decide whether the pair is non-conforming as the safety margin may vary from person to person and from organization to organization.

The following graphs show the scatter chart of the difference from the expected values for all three cases, i.e. single project, Apache and JBoss separately. The 0 includes the rest of the graphs for all projects we have used to validate our model. The equations have resulted very high negative values for some packages, and they have been omitted on the charts to have a better look. The equations have resulted very high negative values for some packages, and they have been omitted on the charts to have a better look.

Similar to the package-level analysis, examining the three cases (single project, Apache and JBoss projects only) separately, we observe that most of the measurements for the expected test metric values is less than or equal to zero, meaning the testing effort for the source class-test class pairs is satisfactory. It is easily seen from the chart that the number of test classes having no test cases are more than the number of packages having no test codes. This implies that these test codes do not contain test cases, but test the related classes with other test statements other than assertments.

For the Apache Ant project, most of the test classes seem to contain more than half of the necessary amount of test lines of code. The number of test cases chart imply that most of the test classes have less than half of the necessary amount of test lines of code. We may state that the testing effort must be increased, especially with more test case assertment statements to have a better testing structure of the test classes. The ratios having a value of one are much in number, meaning that these test classes do not contain assertment statements. For the Apache Lucene project, the majority of the test classes have a divergence ratio less than or equal to zero for all test metrics, meaning test lines of code and number of test cases are adequate.

The Apache Mina, Geronimo and Wicket, JBoss Cache projects indicate similar patterns for both metrics like the Lucene project. JBoss Drools project seems to be the most successful project among the eight old projects in both test metric divergence ratio charts. The majority of the test classes have a divergence ratio less than or equal to zero for dLOC_CLS metric, and most of the classes have a ratio close to or less than zero for dNOTC_CLS metric. Whereas, JBoss Richfaces project seems to be the worst of the 8 projects as half of the ratios for dLOC_CLS metric and most the ratios for dNOTC_CLS metric are between 0 and 1. In addition, the number of test classes with no test cases is most for this project.

Among the new nine projects used to test our model, Apache Tapestry project seems to be the most successful project in LOC test metric's divergence ratio charts. Whereas, this project does not contain any test cases in the test classes. Size of the test classes seems adequate, but the testing is not done by assertment statements, surprisingly.

Apache JackRabbit, OJB, ODE, Meaven, Struts and JBoss ESB indicate successful results for dLOC_CLS metrics, as most of the ratios are either negative or very close to zero. For the dNOTC_CLS metric, Apache JackRabbit shows the most successful results, and most of the projects left indicate a similar pattern to the case of the eight projects, implying that number of test cases should be increased for most of the classes.

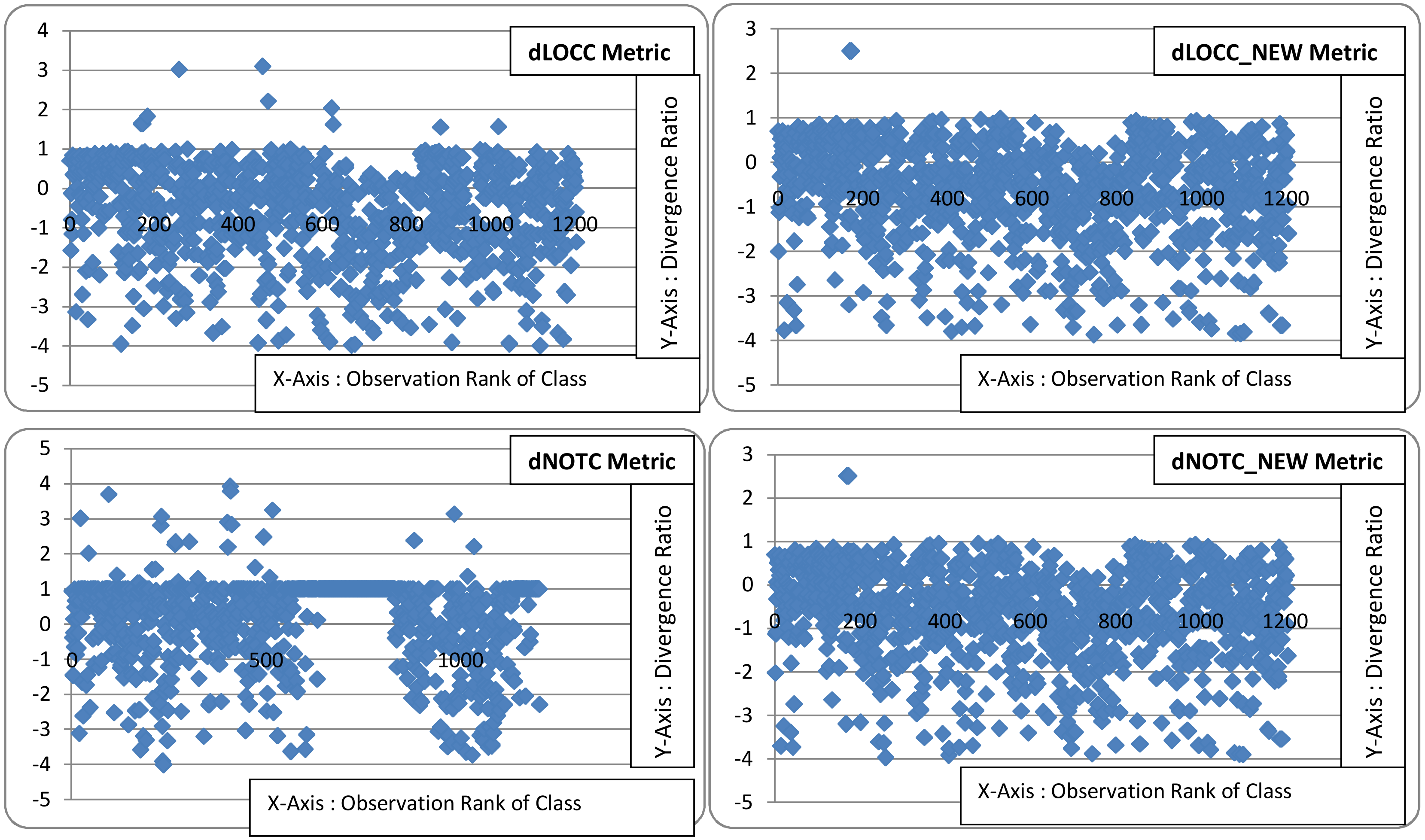


Figure 5 : Scatter Chart of Deviation - Test Metrics - Class Level - All Projects as One Single Project

CHAPTER 7

DISCUSSIONS ON THE MODEL

This chapter provides our discussions on the model we have proposed. Our model primarily addresses to software projects developed within OO environment in Java programming language and tested with JUnit framework.

The class-level and package-level analysis we have performed to evaluate our model in the last section of the previous chapter indicates that our model helps to identify probable non-conforming source class-test class pairs and packages tested less than expected amount. In the majority of the projects we have used in our study successfully, the model produced significant results. Besides determining non-conforming class pairs and packages, it also indicated the pairs and packages tested more than necessary. Examining the divergence ratios and comparing the resulting testing values with the expected values, we may detect the pairs and packages tested more than expected unnecessarily, in other words, spent more testing effort in the test budget than necessary.

Testability is the key concept we try to measure in this study. It is hard to define, measure and explain. Anyone has her own understanding of testability. In the introductory chapters, we had given the two major definitions of testability found in the literature.

IEEE [16] defines testability as “(1) 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. (2) The degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met”. Binder [2] defines the testability of a software system as the relative ease and expense of revealing software faults. These two definitions indicate that the testability of a software system is a direct indicative of the amount of effort, i.e. ease, and cost, i.e. expense needed to test the system.

In chapter 2, we have seen that the major input and output information on the spine of the fishbone (Figure 1 & Figure 2) are the required degree of validity and the required testing effort, consecutively. The required degree of validity defines the level a software project is to be tested. The higher the degree of validity, the higher testing effort is needed, as software that is required to have a high degree of validity will need to be tested thoroughly before it can be claimed the requirement is met.

The required degree of validity varies according to the software project's development purposes and the adhered audience. For example, an embedded software system to be run on a military airplane is a safety-critical system and expected to run with minimum or no error, as safety-critical systems are often required to meet very strict validity requirements; maximally allowable failure rates are typically stated explicitly. On the other hand, a COTS application will not be expected to have the same degree of validity of a safety-critical system.

A software project may have a defined degree of validity or not. If the project has a predefined degree, the testing effort needed is a result of the software development stages and their related aspects, as the goal of the testing is already at the hand. It is not right to expect a required degree of validity at all times. In such cases, it is not straightforward to define the testing effort needed. It will either be according to the available testing effort the software project owner is willing to spend on testing process, or to the defined testing methodology defined in software development process of the project owner.

In common, when a required degree of validity is not defined, the testing effort may be performed depending on some other criterion, which indicates whether necessary testing has been done. An example to such testing criterion is code coverage criterion, common in the context of white box testing, in which the tests rely on information about how the software has been designed or coded. This criterion indicates the extent to which a certain aspect of the code has been "covered" by testing.

In many web-based Java projects, in case of undetermined degree of validity, a certain level of code coverage criterion is defined. For example, the project may be expected to satisfy a minimum of 70% code coverage ratio, defined by the project manager or testing manager. An upper bound is also necessary, as the defined code coverage ratio may require more effort than available resources. Thus, the maximum number of test cases to be generated may be defined as well, to define an upper bound on the testing effort. This means there is a certain trade-off at the testing process due to minimum and maximum constraint.

The testing team will probably have to pay more attention to more critical parts of the software. This raises an important question. Which part of the software do you have to pay more attention? The testing effort is valuable and you have to use your worthy resources in the most effective and efficient manner. Therefore, you have to distribute your testing resources adequately, so that no source is tested more than necessary and less than expected.

Either with a defined degree of validity, or with a certain level of code coverage criterion in case of undetermined degree of validity, you define the maximum effort you want to consume on the related test subject, i.e. class or package. Our model may help to observe whether you have consumed the necessary testing effort. As the two definitions of testability indicate that, the testability of a software system is a direct indicative of the amount of effort, i.e. ease, and cost, i.e. expense needed to test the system, the non-conforming class pairs or packages may be stated to have low testability. You have set a target level, either a degree of validity or a level of code coverage, but the testing effort you have consumed according to one of these criteria seems to be inadequate.

As a result of our model, we define our own testability and state that a class or package in which the same testing criterion, valid all over the project, has been applied and that has a positive value of divergence ratio more than a threshold value defined within the organization has low testability and is hard to test.

Our model measures the testing effort and testability using source-based metrics, meaning that implementation stage must be completed before you may evaluate whether or not you have consumed the adequate testing effort. In an organization, the project staff may review its preliminary or final design taking into account the guidelines proposed by the equations as a natural result of coefficients and their signs.

Our model has used metrics on the most important design parameters. The class and package level expected value equations contain only the major ones of the corresponding design parameters that have significant correlations with testing effort, in other words, insignificant design parameters are omitted from equations as they have ignorable effects. Examining these equations, it is easy to observe which design parameters are crucial in terms of testability and testing effort.

To be able to alter the expected values, one has to increase the corresponding metric value with a negative sign and decrease the metric value with a positive sign. Altering the expected values also lead to testability guidelines for software project staff, as decreasing the expected value means you have spend less amount of effort to test the corresponding class or package, hence increase its testability. For example, to be able to test a package with less testing effort, i.e. smaller expected value of testing metrics, you need to decrease the size of your package, which is obvious, increase the abstractness of you package, decrease the number of interfaces and increase the level of afferent coupling in your package. The examination of the equations indicates strong correlation with the explanations given in the corresponding metric definitions, on to have a high or a low value o the metrics to have a more stable structure of software design.

Considering the purposes of our study and consequences we have obtained evaluating our model, we may summarize that our model helps to:

- define our own understanding of testability,
- observe testability in terms of testing effort,
- identify probable non-conforming source class-test class pairs tested less than expected amount,
- identify probable non-conforming source packages tested less than expected amount,
- identify the major source based metrics affecting the testing effort,
- identify the major design parameters affecting the testing effort,
- define guidelines to alter testability level,
- perform and evaluate software design according to testability and testing effort,

Our study focused on Object-Oriented Software Systems only. The reason why we have focused is that OO programming is a popular and commonly used programming paradigm, which has not been examined with respect to older paradigms. In addition, OO software systems are widely used in web-based systems, which provide easy data measurement facility, as well.

Many new trends are becoming popular in software development discipline. Some of these new trends and approaches are:

- Component-Oriented Software Development
- Aspect-Oriented Software Development
- Agile Development

Our model may be applicable in these and other new trends, as well. Some of the terms that compound the fundamentals of the software development environment may alter, such as aspects take place of objects in Aspect-Oriented Software Development. Nevertheless, most of the basic concepts and principles of software development that we have based our model onto are general and not peculiar to OO environment only.

We have used the open-source projects written in Java language belonging to two popular open-source frameworks, Apache and JBoss. The systems of these two organizations that are subject to our research are unit tested at the class level using the JUnit testing framework [47]. Detailed inspection of the software projects and leading organizations of these open-source frameworks indicate that they do not act different that commercial software companies. The major difference from a commercial company is that the resulting product does not aim to produce revenue (although in some cases and licensing models, it may) and the product is open to public, both in terms of usage and development. Therefore, it is crucial to note that our model is definitely applicable in commercial software projects, as well.

Under normal conditions, the expected test effort equations and the proposed new test metrics would normally be very close to the proposed ones in case of different study data and scope. Because, the number of projects examined and the mathematical significances of nearly all computations are very high and our data set contains a strong and a successful representation of our domain. The projects belong to popular open-source frameworks, Apache and JBoss. Due to contribution of many different people in many different projects, we may evaluate open-source frameworks as a huge organization or project, containing many different and mostly independent, sub-projects or organizations. From mathematical perspective, the number of projects we have used in our studies are significant enough not to differ by adding other projects to our data set.

CHAPTER 8

CONCLUSIONS

This final chapter provides the concluding remarks on our research. It summarizes the study, presents the contributions performed by our model and research and defines the future work to be performed.

8.1 Conducted Study

The primary concern of this dissertation, as stated at the beginning of this work, is the factors that affect testability and testing cost of object-oriented software systems. Testing is an important software development activity with a primary purpose of detecting the errors in a software program. This process consumes a significant amount of time and effort within an average software development project.

There have been numerous research efforts and studies embracing the importance of testing and the derived benefits. Since software testing is so important, we should not see it as a separate process that takes place close to the end of the development life cycle. Developing a software program, which eases the testing process by increasing testability, is crucial.

To be able to assess whether or not the testing effort and cost consumed is adequate is another critical matter this dissertation aimed to answer by composing new way to evaluate the links between software design parameters and testing effort via source-based metrics. Software projects belonging to two different open-source frameworks helped us to achieve our goals.

Our study used five projects from Apache organization and three projects from JBoss with similar size in lines of code. As the number of projects differ for the two organization, their effects also differ on the case of considering all projects as one single project according to their code size. The main reason Apache took place more in number of projects is that Apache projects had more testing code in JUnit framework [47] embedded in the source code releases.

In our dissertation, we have presented significant associations, relationships and properties of source based metrics in many different levels, i.e. method, class, package and project. We have proposed new test metrics in various levels. We have found significant associations between the source-based metrics and the test suite metrics. We have also examined the relationships among the source-based metrics, as well to observe how different metrics belonging to different design parameters affect each other.

We have also performed regression analysis in both class and package levels, and proposed new equations for obtaining the expected metric values so that we can conclude that the packages are adequate, and source class-test class pair has a healthy relationship to be tested properly and necessarily.

We have composed a new model of testing effort and testability via the proposed equations using the available object-oriented software metrics. The new model we have proposed is significant, as there are only a few models in the literature proposed on testing effort and testability concept. We have tested our model on new open-source projects, which have not been used in any part of our study. The results of testing our model validated the strength and success of our model to define expected values for the test metrics, which help us to identify probable non-conforming testing components (packages or test class pairs) in our project.

8.2 Contributions

Considering the purposes of our study and consequences we have obtained evaluating and testing our model, as defined in CHAPTER 6, we may summarize that our model helps to:

- define our own understanding of testability,
- observe testability in terms of testing effort,
- identify probable non-conforming source class-test class pairs tested less than expected amount,
- identify probable non-conforming source packages tested less than expected amount,

- identify the major source based metrics affecting the testing effort,
- identify the major design parameters affecting the testing effort,
- define guidelines to alter testability level,
- perform and evaluate software design according to testability and testing effort.

Our model defines mathematical equations for obtaining the expected values of the test metrics, in two different levels, i.e. class and package levels. The guidelines stated in 5.2 helps one to make use of our model and equations in two different points of view: either during design phase, i.e. or after the implementation phase.

Considering the use of the model in the design phase of the project, to be able to alter the expected values, one has to increase the corresponding metric value with a negative sign and decrease the metric value with a positive sign. The examination of the equations indicates strong correlation with the explanations given in the corresponding metric definitions, on to have a high or a low value o the metrics to have a more stable structure of software design

Making use of our model and equations after the implementation phase is completed means that one aims to identify probable non-conforming source-test pairs, which indicate that the testing effort is not adequate for these pairs. In order to be able to make use of our model and equations after the implementation phase is completed, one has to follow the steps stated in 5.2, which are similar to the steps we have used to construct and validate our model.

The approach presented in this dissertation brings a number of essential contributions to the field of testing effort and testability assessment in object-oriented design based on software metrics. These contributions are summarized below as follows:

- We used eight open-source projects to compose our model. Nine more similar projects from the same organizations were used in validation of the model. We have composed three different groupings to define a greater picture to see the effect of the context on the relationships we have examined throughout our study. These three groupings were considering all eight projects as one single project, and grouping Apache and JBoss projects only, separately.
- We performed a statistical evaluation of the metrics and presented significant associations, relationships and properties of source based metrics in many different levels, i.e. method, class, package and project levels.
- We have proposed new test metrics in various levels. We have found significant associations between the source-based metrics and the test suite metrics. We have also

examined the relationships among the source-based metrics, as well to observe how different metrics belonging to different design parameters affect each other.

- New metrics proposed in the class-level (dLOC_CLS_NEW and dNOTC_NEW_CLS) turn out to have nearly perfect correlation among each other with a coefficient of 0.99. This implies that one of them is enough to examine test class characteristics.
- These two new test class metrics proposed produce very close expected model values as a result of this high correlation, although their values are computed using different metric set with different coefficients in the equations.
- We have proposed new equations for obtaining the expected metric values so that we can conclude that the packages are adequate, and source class-test class pair has an promising relationship to be tested properly and necessarily.
- We have proposed a new divergence ratio to compare observed and expected metric values.
- We have composed a new model of testing effort and testability via the proposed equations using the available object-oriented software metrics. The new model we have proposed is significant, as there are only a few models in the literature proposed on testing effort and testability concept.
- We have validated our model on new open-source projects, which have not been used in the model construction part of our study. The results of testing our model supported the strength and success of our model to define expected values for the test metrics, which help us to identify probable non-conforming testing components (packages or test class pairs) in our project.
- We have stated our own view of testability concept and approach through testing effort, by evaluating the output of our model, the expected testing effort equations.

8.3 Future Work

Below are given the steps we aim to perform in the future to expand and strengthen our model.

- In order to expand the perspective of testability and testing effort used in this dissertation, the projects of larger software development organizations, both commercial and open-source, may be used to extend the number and variety of projects to compose our model onto. If available, the bug database of these projects may be used to map a relationship between testability and source metrics, similar to the NASA approach, with stronger fundamentals.

    - The validity of our results and model could be explored for systems written in other object-oriented languages, such as C++, DOT.NET, as the definitions of our metrics contain only a limited amount of dependency on the programming language of our case studies.
    - Our model could be validated with different projects other than the organizations, whose projects have already been the fundamental projects used to compose our results and equations.
    - If possible, our model could be validated with different projects having testing frameworks other than the JUnit testing framework [47].
- The metrics tool could be extended to contain many other object-oriented metrics that have not been taken into account in the current releases of plug-in. New test metrics could also be proposed, as well, to better identify the relations between source and test components, i.e. class, package and project levels.
- Our model could be revisited and revised using different non-parametric statistics other than the primary mathematical function used in our study, Spearman's rank-order correlation coefficient [40].
- Many other complicated forms of regression, such as polynomial, logarithmic, exponential equations, may be applied instead of the linear regression model we have used in our study, which simplest form of regression, but chosen due to calculation simplicity and availability.
- Our model may be applied in new trends and approaches in software development, such as Component-Oriented, Aspect-Oriented and Agile Development methodologies, as well. Some of the terms that compound the fundamentals of the software development environment may be different, in these new approached, but it must be taken into account that most of the basic concepts and principles of software development that we have based our model onto are general and not peculiar to OO environment only.

BIBLIOGRAPHY

1. Beck K., Test-Driven Development : By Example, Addison-Wesley, 2002.

2. Binder R.V., “Desing for Testability in OO Systems”, Communications of the ACM, Sep. 1994.

3. Boehm B. W., Brown J. R., Kaspar H., Lipow M., Macleod G. J. and Merritt M. J., “Characteristics of Software Quality”, Amsterdam: North-Holland, 1978.

4. Booch G., Object-Oriented Analysis and Design with Applications, 2nd edition, Benjamin Cummings, Redwood City, 1994.

5. Borchert T., “Code Profiling : Static Code Analysis”, E-level thesis, Karlstads Universitet, 2008.

6. Briand L. C., Morasca S., and Basili V., “An operational process for goal-driven definition of measures”, IEEE Transactions on Software Engineering, 28(12):1106-1125, December 2002.

7. Bruntink M., “Testability of Object-Oriented Systems: a Metrics-based Approach”, Master's Thesis, Universiteit van Amsterdam, September 2003.

8. Budd T., An Introduction to OO Programming, 3rd Edition, Addison-Wesley, 2003.

9. Chidamber S. and Kemerer C., “A metrics suite for object oriented design”, IEEE Transactions on Software Engineering, 20(6):476-493, June 1994.

10. Da-wei E., “The Software Complexity Model and Metrics for Object-Oriented”, 2007 IEEE International Workshop on Anti-counterfeiting, Security, Identification: 464-469, April 2007.

11. Dromey R., “Comering the Chimera”, 13 (1): 33-43, IEEE Software, January 1996.

12. Eclipse metrics plug-in (v 1.3.6) by Frank Sauer, http://metrics.sourceforge.net.

13. Gamma E., Helm R., Johnson R., Vlissides J., “Design Patterns Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1994.

14. Halstead M. H., “Elements of Software Science, Operating, and Programming Systems Series”, Volume 7, New York, NY: Elsevier, 1977.

15. Henderson-Sellers B., Object-Oriented Metrics: Measures of Complexity, Prentice Hall, 1995.

16. IEEE standard glossary of software engineering terminology, ISBN-13: 978-1559370677, IEEE, 1991.

17. Jungmayr S., “Design for Testability ”, CONQUEST 2002, September 2002.

18. Karlsson E., “Software reuse, a holistic approach”, John Wiley & Sons Inc, 1995.

19. Kent A., Williams J.G., Kent R., Hall C.M., Marciniak J. (Editor-in-chief), Encyclopedia of software engineering, Wiley-Interscience, 1994.

20. Khan R. A. and Mustafa K., “A Model For Object Orıented Design Quality Assessment”, Jamia Millia University, 2004.

21. Lorenz M. and Kidd J.. “Object-Oriented Software Metrics”. Prentice-Hall Object-Oriented Series, Englewood Cliffs, NY, 1994.

22. Losaviol F., Chirinos L. and Pérez M., “Quality Models to Design Software Architectures. Technology of Object-Oriented Languages and Systems”, In Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS 2001), pp: 123–135, 2001

23. Marinescu R., “Measurement and Quality in Object-Oriented Design”, Ph.D. Thesis, University of Timişoara, October 2002.

24. Martin R., “Agile Software Development, Principles, Patterns and Practices”, Prentice Hall; 1st edition, 2002.

25. Martin R., “OO Design Quality Metrics, An Analysis of Dependencies”, In Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA'94, 1994.

26. McCall J. A., Richards P. G. and Walters G. F., “Factors in Software Quality”, Springfield: NTIS, 1977.

27. McConnell S., Code Complete, 2nd edition, Microsoft Press, 2004.

28. Mills E., “Software Metrics”, SEI Curriculum Module SEI-CM-12-1.1, December 1988.

29. Mitchell J.C., Concepts In Programming Languages, Cambridge University Press, 2003.

30. NASA IV&V Facility Metrics Data Program, http://mdp.ivv.nasa.gov.

31. Object Technology International, Inc. Eclipse Platform Technical Overview, February 2003, http://www.eclipse.org.

32. Siegel S. and Castellan N.J., “Nonparametric statistics for the behavioral sciences”, McGraw-Hill Book Company, New York, 1988.

33. Software Engineering Body of Knowledge (SWEBOK), Software Engineering Coordinating Committee, 2004, http://www.swebok.org.

34. Sommerville I., Software Engineering, 7th Edition, Addison-Wesley, 2004.

35. Sun Microsystems, Java Coding Standard, http://java.sun.com/docs/codeconv.

36. Voas J., Morell L., and Miller K.. “Predicting where faults can hide from testing”. IEEE Software, 8:41-48, March 1991.

37. Watson A. and McCabe T., "Structured testing: A software testing methodology using the cyclomatic complexity metric”, National Institute of Standards and Technology Special Publication 500-235, 1996.

38. Wheeldon R. and Counsell S., “Power law distributions in class relationships”, In Proceedings of the Third International Workshop on Source Code Analysis and Manipulation, IEEE Computer Society, September 2003.

39. Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Software_testing [1 February 2009].

40. Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Spearman's_rank_correlation_coefficient [1 February 2009].

41. Xenos M., Stavrinoudis D., Zikouli K. and Christodoulakis D., "Object Oriented Metrics - A survey", Proceedings of the FESMA 2000, Federation of European Software Measurement Associations, Madrid, Spain, 2000.

42. Xie T., Huang H., Chen X., Mei H. and Yang F., "Object Oriented Software Metrics Technology - Technical Report", Peking University, 2001.

43. Glasberg D., Emam K., Melo W., Madhavji N., "Validating Object-Oriented Design Metrics on a Commercial Java Application", National Research Institute for Information Technology, 2000.

44. The Apache Software Foundation, http://www.apache.org.

45. JBoss Community, http://www.jboss.org.

46. Red Hat Company, http://www.redhat.com.

47. JUnit Testing Framework, http://www.junit.org.

48. Oracle 11g Database Management System, http://www.oracle.com.

APPENDICES

Statistical Results of the Apache-JBoss Project Repository & Definitions of Software Design Paramaters

APPENDIX A. Inter-Metrics Correlation Analysis Details

Table 32 : Significance Values Among Package Metrics : Correlation Analysis – All As One Single Project

	CA	***CE***	***LOC_PKG***	***NOC***	***NOI***	***RMA***	***RMD***	***RMI***
CA	0							
CE	0	0						
LOC_PKG	0	0	0					
NOC	0	0	0	0				
NOI	0	0	2.52E-83	2.83E-93	0			
RMA	0	2.20E-48	2.52E-18	2.74E-09	0	0		
RMD	0	0.121115	7.93E-21	6.33E-39	1.56E-45	1.34E-41	0	
RMI	0	0.002926	1.87E-21	3.52E-05	0	0	0	0

Table 33 : Correlation Values Among Package Metrics– Apache Projects Only

	CA	***CE***	***LOC_PKG***	***NOC***	***NOI***	***RMA***	***RMD***	***RMI***
CA	1.00							
CE	0.40	1.00						
LOC_PKG	0.37	0.71	1.00					
NOC	0.35	0.84	0.80	1.00				
NOI	0.52	0.35	0.29	0.26	1.00			
RMA	0.45	0.20	0.12	0.06	0.75	1.00		
RMD	0.60	0.03	0.13	0.16	0.16	0.13	1.00	
RMI	-0.86	0.01	-0.09	-0.02	-0.39	-0.40	-0.70	1.00

Table 34 : Significance Values Among Package Metrics : Correlation Analysis – Apache Projects Only

	CA	***CE***	***LOC_PKG***	***NOC***	***NOI***	***RMA***	***RMD***	***RMI***
CA	0							
CE	1.85E-103	0						
LOC_PKG	8.94E-84	0	0					
NOC	1.30E-77	0	0	0				
NOI	0	3.43E-78	2.31E-52	3.02E-42	0			
RMA	0	1.12E-25	3.05E-09	0.001373	0	0		
RMD	0	0.136977	9.92E-11	9.77E-17	4.3E-16	1.2E-11	0	
RMI	0	0.792435	3.32E-06	0.249025	1.50E-98	0.0001	0	0

Table 35 : Correlation Values Among Package Metrics : Correlation Analysis – JBoss Projects Only

	CA	***CE***	***LOC_PKG***	***NOC***	***NOI***	***RMA***	***RMD***	***RMI***
CA	1.00							
CE	0.23	1.00						
LOC_PKG	0.30	0.66	1.00					
NOC	0.27	0.76	0.71	1.00				
NOI	0.32	0.23	0.17	0.22	1.00			
RMA	0.33	0.13	0.07	0.06	0.81	1.00		
RMD	0.62	0.00	0.09	0.15	0.17	0.18	1.00	
RMI	-0.90	0.09	-0.10	-0.04	-0.26	-0.30	-0.68	1.00

Table 36 : Significance Values Among Package Metrics : Correlation Analysis – JBoss Projects Only

	CA	***CE***	***LOC_PKG***	***NOC***	***NOI***	***RMA***	***RMD***	***RMI***
CA	0							
CE	2.95E-51	0						
LOC_PKG	8.11E-87	0	0					
NOC	2.40E-68	0	0	0				
NOI	2.86E-101	1.50E-51	1.02E-28	1.67E-48	0			
RMA	7.42E-105	3.21E-18	5.35E-06	0.000218	0	0		
RMD	0	0.827142	2.41E-09	5.35E-22	7.07E-30	2.04E-30	0	
RMI	0	1E-09	8.72E-10	0.006807	5.27E-65	4.70E-90	0	0

Table 37 : Significance Values Among Method Metrics : Correlation Analysis – All As One Single Project

	MLOC	***NBD***	***PAR***	***VG***
MLOC	0			
NBD	0	0		
PAR	0	0	0	
VG	0	0	0	0

Table 38 : Significance Values Among Class Metrics : Correlation Analysis – All As One Single Project

	DIT	***FOUT***	***LCOM***	***LOCC***	***NOF***	***NOM***	***NORM***	***NSC***	***NSF***	***NSM***	***RFC***	***SIX***	***TNOF***	***TNOM***	***LOCC_NEW***	***WMC***
DIT	0															
FOUT	7.2E-71	0														
LCOM	3.2E-26	0	0													
LOCC	1.4E-44	0	0	0												
NOF	1.3E-24	0	0	0	0											
NOM	2.1E-63	0	0	0	0	0										
NORM	0	0	0	0	0	0	0									
NSC	5.3E-36	3.3E-58	8E-103	4E-88	5E-103	0	1.2E-37	0								
NSF	0.0001	0	0	0	0	0	6E-75	1E-17	0							
NSM	5.2E-09	0	3.1E-38	0	7E-77	2.89E-116	2E-16	0.0008	0	0						
RFC	0	0	0	0	0	0	0	3E-128	0	0	0					
SIX	0	0	0	0	0	0	0	7E-27	1E-48	2E-20	0	0				
TNOF	3.2E-35	0	0	0	0	0	0	8E-83	0	2E-40	0	0	0			
TNOM	6.7E-19	0	0	0	0	0	0	0	0	0	0	0	0	0		
LOCC_NEW	1.3E-42	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
WMC	0.005	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 39 : Correlation Values Among Class Metrics : Correlation Analysis – Apache Projects Only

	DIT	***FOUT***	***LCOM***	***LOCC***	***NOF***	***NOM***	***NORM***	***NSC***	***NSF***	***NSM***	***RFC***	***SIX***	***TNOF***	***TNOM***	***WMC***
DIT	1.00														
FOUT	0.03	1.00													
LCOM	-0.10	0.34	1.00												
LOCC	0.00	0.79	0.48	1.00											
NOF	-0.12	0.39	0.87	0.51	1.00										
NOM	0.04	0.44	0.67	0.64	0.69	1.00									
NORM	0.30	0.22	0.16	0.20	0.20	0.32	1.00								
NSC	0.05	0.08	0.09	0.09	0.10	0.19	0.06	1.00							
NSF	0.02	0.33	0.17	0.33	0.17	0.20	0.09	0.05	1.00						
NSM	-0.09	0.30	-0.03	0.28	-0.05	-0.08	-0.04	-0.01	0.28	1.00					
RFC	0.07	0.83	0.50	0.88	0.52	0.73	0.28	0.12	0.31	0.25	1.00				
SIX	0.33	0.18	0.11	0.16	0.15	0.26	0.98	0.05	0.07	-0.05	0.23	1.00			
TNOF	-0.12	0.45	0.77	0.56	0.87	0.62	0.18	0.09	0.55	0.10	0.56	0.13	1.00		
TNOM	-0.02	0.53	0.64	0.74	0.64	0.93	0.29	0.17	0.28	0.17	0.82	0.23	0.63	1.00	
WMC	-0.06	0.72	0.57	0.91	0.60	0.79	0.26	0.14	0.33	0.25	0.89	0.20	0.63	0.89	1.00

Table 40 : Significance Values Among Class Metrics : Correlation Analysis – Apache Projects Only

	DIT	***FOUT***	***LCOM***	***LOCC***	***NOF***	***NOM***	***NORM***	***NSC***	***NSF***	***NSM***	***RFC***	***SIX***	***TNOF***	***TNOM***	***WMC***
DIT	0														
FOUT	9.31E-05	0													
LCOM	3.67E-52	0	0												
LOCC	0.84	0	0	0											
NOF	1.6E-64	0	0	0	0										
NOM	6.8E-09	0	0	0	0	0									
NORM	0	0	5.7E-118	0	0	0	0								
NSC	4.5E-14	3E-34	7.9E-42	8.68E-38	2E-45	0	1.4E-19	0							
NSF	0.02	0	1.1E-129	0	0	0	2.10E-37	3.02E-11	0						
NSM	2E-39	0	0.0001	0	2.7E-13	7.89E-31	1E-08	0.31	0	0					
RFC	2.2E-24	0	0	0	0	0	0	1.34E-72	0	0	0				
SIX	0	0	3.9E-55	6E-118	1E-103	0	0	5.9E-14	1.64E-21	2.70E-12	0	0			
TNOF	2.3E-70	0	0	0	0	0	0	3.68E-36	0	1.60E-50	0	6.75E-80	0		
TNOM	0.027	0	0	0	0	0	0	0	0	0	0	0	0	0	
WMC	1E-17	0	0	0	0	0	0	2.90E-95	0	0	0	0	0	0	0

Table 41 : Correlation Values Among Class Metrics : Correlation Analysis – JBoss Projects Only

	DIT	***FOUT***	***LCOM***	***LOCC***	***NOF***	***NOM***	***NORM***	***NSC***	***NSF***	***NSM***	***RFC***	***SIX***	***TNOF***	***TNOM***	***WMC***
DIT	1.00														
FOUT	0.13	1.00													
LCOM	0.00	0.26	1.00												
LOCC	0.11	0.82	0.41	1.00											
NOF	0.01	0.30	0.84	0.45	1.00										
NOM	0.12	0.38	0.61	0.59	0.66	1.00									
NORM	0.32	0.21	0.22	0.24	0.25	0.34	1.00								
NSC	0.06	0.07	0.11	0.10	0.10	0.17	0.06	1.00							
NSF	0.03	0.31	0.11	0.33	0.13	0.16	0.09	0.04	1.00						
NSM	0.03	0.31	-0.09	0.28	-0.12	-0.14	-0.04	-0.02	0.22	1.00					
RFC	0.18	0.82	0.47	0.86	0.50	0.71	0.30	0.10	0.27	0.24	1.00				
SIX	0.35	0.18	0.18	0.21	0.21	0.29	0.99	0.05	0.08	-0.04	0.26	1.00			
TNOF	0.00	0.38	0.73	0.52	0.84	0.58	0.23	0.09	0.56	0.03	0.53	0.19	1.00		
TNOM	0.10	0.47	0.59	0.69	0.61	0.93	0.31	0.16	0.23	0.10	0.80	0.26	0.60	1.00	
WMC	0.08	0.66	0.52	0.88	0.56	0.80	0.30	0.14	0.31	0.18	0.87	0.25	0.60	0.89	1.00

Table 42 : Significance Values Among Class Metrics : Correlation Analysis – JBoss Projects Only

	DIT	***FOUT***	***LCOM***	***LOCC***	***NOF***	***NOM***	***NORM***	***NSC***	***NSF***	***NSM***	***RFC***	***SIX***	***TNOF***	***TNOM***	***WMC***
DIT	0														
FOUT	5.84E-81	0													
LCOM	0.89	0	0												
LOCC	7.06E-58	0	0	0											
NOF	0.04	0	0	0	0										
NOM	3.14E-60	0	0	0	0	0									
NORM	0	0	0	0	0	0	0								
NSC	5.4E-20	1E-23	5E-54	2E-43	1E-48	0	1.6E-17	0							
NSF	1.15E-06	0	6E-57	0	6E-71	1E-117	7.8E-41	2E-08	0						
NSM	1.80E-06	0	8E-39	0	1E-69	8E-90	1E-08	0.007	0	0					
RFC	0	0	0	0	0	0	0	9.5E-48	0	0	0				
SIX	0	0	0	0	0	0	0	1.9E-12	2E-30	2.6E-09	0	0			
TNOF	0.54	0	0	0	0	0	0	1.9E-40	0	1.5E-05	0	0	0		
TNOM	1E-41	0	0	0	0	0	0	6E-114	0	6E-49	0	0	0	0	
WMC	1E-27	0	0	0	0	0	0	7.4E-91	0	0	0	0	0	0	0

Table 43 : Regression Analysis for Test Metrics Values – Apache Only– Package Level

dLOC-PKG				**dNOTC-PKG**		
	Coefficients	*P-value*			*Coefficients*	*P-value*
CA	**-0.8259**	0.0624		**CA**	-0.0713	0.3476
CE	**-16.3202**	0.0001		**CE**	**-2.8759**	7.47344E-05
NOC	**4.7955**	0.0291		**NOC**	**0.7457**	0.0480
NOI	**105.2167**	4.86131E-16		**NOI**	**15.1466**	4.96226E-12
RMA	**-1709.4947**	3.50577E-07		**RMA**	**-234.2641**	4.18814E-05
RMD	63.1122	0.5443		**RMD**	25.9757	0.1465
RMI	42.0161	0.4800		**RMI**	12.0476	0.2383
LOC-PKG	**0.3185**	0.0000		**LOC-PKG**	**0.0463**	1.01638E-15

Table 44 : Regression Analysis for Test Metrics Values – JBoss Only– Package Level

dLOC-PKG				**dNOTC-PKG**		
	Coefficients	*P-value*			*Coefficients*	*P-value*
CA	-0.7902	0.5271		**CA**	0.0310	0.9096
CE	**46.9857**	1.54403E-05		**CE**	**6.2292**	0.0072
NOC	**-21.8011**	0.0054		**NOC**	**-7.5888**	1.57837E-05
NOI	22.4567	0.4550		**NOI**	3.3226	0.6133
RMA	-778.3241	0.3216		**RMA**	-93.7194	0.5850
RMD	272.8416	0.1923		**RMD**	**75.6809**	0.0993
RMI	55.2756	0.6509		**RMI**	**47.9868**	0.0746
LOC-PKG	**0.1871**	0.0013		**LOC-PKG**	**0.0827**	1.09774E-09

Table 45 : Correlation Between Expected Regression Test Values : All Projects as One Single Project Case versus JBoss Projects Only Case – Package Level

CORRELATION VALUES		*SINGLE PROJECT*	
		dLOC_PKG	dNOTC_PKG
JBOSS ONLY	dLOC_PKG	**0.746**	0.695
	dNOTC_PKG	0.421	**0.469**

Table 46 : Correlation Between Expected Regression Test Values : All Projects as One Single Project Case versus Apache Projects Only Case – Package Level

CORRELATION VALUES		*SINGLE PROJECT*	
		dLOC_PKG	dNOTC_PKG
APACHE ONLY	dLOC_PKG	**0.986**	0.988
	dNOTC_PKG	0.975	**0.996**

Table 47 : Correlation Between Expected Regression Test Values : JBoss Projects Only Case versus Apache Projects Only Case – Package Level

CORRELATION VALUES		*APACHE ONLY*	
		dLOC_PKG	dNOTC_PKG
JBOSS ONLY	dLOC_PKG	**0.635**	0.611
	dNOTC_PKG	0.361	**0.355**

Table 48 : Correlation Between Expected Regression Test Values : All Projects as One Single Project Case versus JBoss Projects Only Case – Class Level

CORRELATION VALUES		***SINGLE PROJECT***			
		dLOC_CLS	**dLOC_NEW**	**dNOTC**	**dNOTC_NEW**
JBOSS ONLY	**dLOC_CLS**	**0.676**	0.769	0.764	0.771
	dLOC_NEW	0.744	**0.742**	0.673	0.744
	dNOTC	0.744	0.850	**0.849**	0.852
	dNOTC_NEW	0.744	0.746	0.676	**0.747**

Table 49 : Correlation Between Expected Regression Test Values : All Projects as One Single Project Case versus Apache Projects Only Case – Class Level

CORRELATION VALUES		***SINGLE PROJECT***			
		dLOC_CLS	**dLOC_NEW**	**dNOTC**	**dNOTC_NEW**
APACHE ONLY	**dLOC_CLS**	**0.857**	0.911	0.653	0.909
	dLOC_NEW	0.890	**0.958**	0.724	0.957
	dNOTC	0.890	0.763	**0.758**	0.763
	dNOTC_NEW	0.890	0.959	0.725	**0.958**

Table 50 : Correlation Between Expected Regression Test Values : JBoss Projects Only Case versus Apache Projects Only Case – Class Level

CORRELATION VALUES		***APACHE ONLY***			
		dLOC_CLS	**dLOC_NEW**	**dNOTC**	**dNOTC_NEW**
JBOSS ONLY	**dLOC_CLS**	**0.625**	0.763	0.843	0.764
	dLOC_NEW	0.536	**0.654**	0.570	0.656
	dNOTC	0.646	0.774	**0.778**	0.776
	dNOTC_NEW	0.539	0.658	0.577	**0.660**

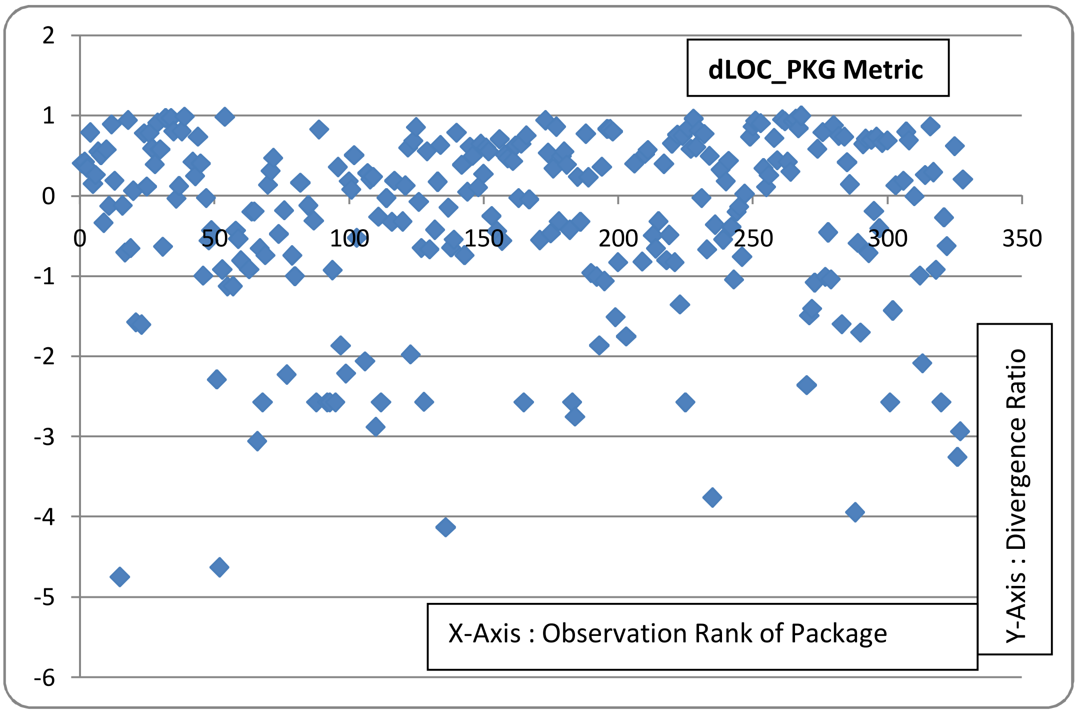
Table 51 : Regression Analysis for Test Metrics Values - Apache Projects Only Case – Class Level

dLOC_CLS				**dLOCC_NEW**				**dNOTC**				**dNOTC_NEW**		
	Coefficients	***P-value***			***Coefficients***	***P-value***			***Coefficients***	***P-value***			***Coefficients***	***P-value***
DIT	***3.7886***	0.0075		**DIT**	***2.7906***	0.0018		**DIT**	0.4741	0.1245		**DIT**	***1.9409***	0.0018
FOUT	***3.3077***	7.4E-05		**FOUT**	***2.8252***	8.92E-08		**FOUT**	***0.4608***	0.0112		**FOUT**	***1.9665***	8.2E-08
LCOM	***19.2434***	0.0394		**LCOM**	***13.4271***	0.0226		**LCOM**	2.0541	0.3128		**LCOM**	***9.2691***	0.0234
LOCC	0.0465	0.5087		**LOCC**	-0.0174	0.6945		**LOCC**	0.0061	0.6924		**LOCC**	-0.0126	0.6828
NOF	0.0000	0		**NOF**	0.0000	0.0000		**NOF**	0.0000	0.0000		**NOF**	0.0000	0.0000
NOM	0.2093	0.7793		**NOM**	-0.0159	0.9731		**NOM**	-0.1944	0.2327		**NOM**	-0.0170	0.9584
NORM	3.8598	0.1636		**NORM**	2.8540	0.1023		**NORM**	0.7483	0.2154		**NORM**	1.9761	0.1031
NSC	***0.9178***	0.0966		**NSC**	***0.6142***	0.0777		**NSC**	-0.0145	0.9040		**NSC**	***0.4218***	0.0810
NSF	1.5105	0.1232		**NSF**	0.8419	0.1728		**NSF**	***0.8414***	8.6E-05		**NSF**	0.5933	0.1663
NSM	0.0000	0		**NSM**	0.0000	0.0000		**NSM**	0.0000	0		**NSM**	0.0000	0.0000
RFC	***-1.3683***	6.5E-07		**RFC**	***-0.8616***	0.0000		**RFC**	***-0.1910***	0.0014		**RFC**	***-0.5961***	7.2E-07
SIX	-4.8731	0.4373		**SIX**	-3.1755	0.4220		**SIX**	-2.0576	0.1327		**SIX**	-2.2258	0.4174
TNOF	-0.5335	0.5342		**TNOF**	-0.1317	0.8077		**TNOF**	-0.9065	1.4E-06		**TNOF**	-0.1008	0.7883
TNOM	0.6712	0.4080		**TNOM**	0.4263	0.4044		**TNOM**	***0.5172***	0.0035		**TNOM**	0.3069	0.3872
WMC	***1.0118***	0.0001		**WMC**	***0.7697***	4.33E-06		**WMC**	***0.1635***	0.0046		**WMC**	***0.5346***	4.2E-06

Table 52 : Regression Analysis for Test Metrics Values - JBoss Projects Only Case – Class Level

dLOC_CLS			dLOC_CLS_NEW			dNOTC			dNOTC_NEW		
	Coefficients	***P-value***		***Coefficients***	***P-value***		***Coefficients***	***P-value***		***Coefficients***	***P-value***
DIT	**19.8064**	4.87E-07	**DIT**	**12.7824**	1.65E-08	**DIT**	**1.8635**	0.0327	**DIT**	**8.8901**	1.8E-08
FOUT	-0.8761	0.6301	**FOUT**	-0.0872	0.9331	**FOUT**	-0.3396	0.4094	**FOUT**	-0.0792	0.9131
LCOM	1.8006	0.9313	**LCOM**	9.8458	0.4098	**LCOM**	-4.3959	0.3526	**LCOM**	6.7176	0.4206
LOCC	0.2198	0.1149	**LOCC**	0.1188	0.1359	**LOCC**	**0.0652**	0.0390	**LOCC**	0.0833	0.1340
NOF	**-7.0066**	0.0215	**NOF**	**-4.3939**	0.0117	**NOF**	**-1.7438**	0.0115	**NOF**	**-3.0682**	0.0117
NOM	**6.1283**	0.0933	**NOM**	3.3213	0.1114	**NOM**	0.5719	0.4877	**NOM**	2.3098	0.1128
NORM	-4.6490	0.4011	**NORM**	-4.5787	0.1484	**NORM**	1.3964	0.2651	**NORM**	-3.2067	0.1472
NSC	3.0069	0.5567	**NSC**	2.7961	0.3391	**NSC**	-0.4330	0.7082	**NSC**	1.8828	0.3565
NSF	0.0000	0.0000	**NSF**	0.0000	0.0000	**NSF**	0.0000	0.0000	**NSF**	0.0000	0.0000
NSM	0.0000	0.0000	**NSM**	0.0000	0.0000	**NSM**	0.0000	0.0000	**NSM**	0.0000	0.0000
RFC	**2.6264**	0.0003	**RFC**	**1.1764**	0.0048	**RFC**	**0.4387**	0.0078	**RFC**	**0.8379**	0.0040
SIX	**-27.6478**	0.0999	**SIX**	-13.8804	0.1481	**SIX**	**-8.6230**	0.0236	**SIX**	-9.7161	0.1470
TNOF	0.1188	0.9612	**TNOF**	-0.0032	0.9982	**TNOF**	0.5056	0.3603	**TNOF**	0.0104	0.9915
TNOM	-5.5533	0.1359	**TNOM**	-2.3614	0.2667	**TNOM**	-0.6562	0.4353	**TNOM**	-1.6440	0.2681
WMC	-0.6325	0.4022	**WMC**	-0.2943	0.4950	**WMC**	0.0429	0.8013	**WMC**	-0.2152	0.4749

APPENDIX B. Package Test Metrics - Scatter Charts of Deviation From Expected



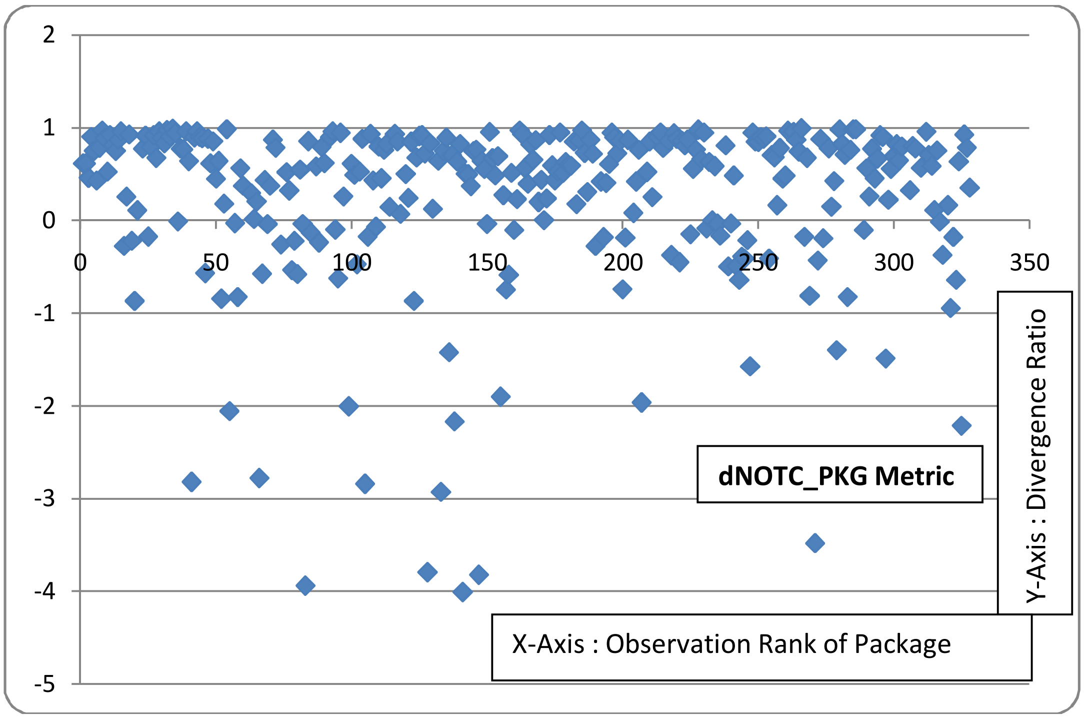
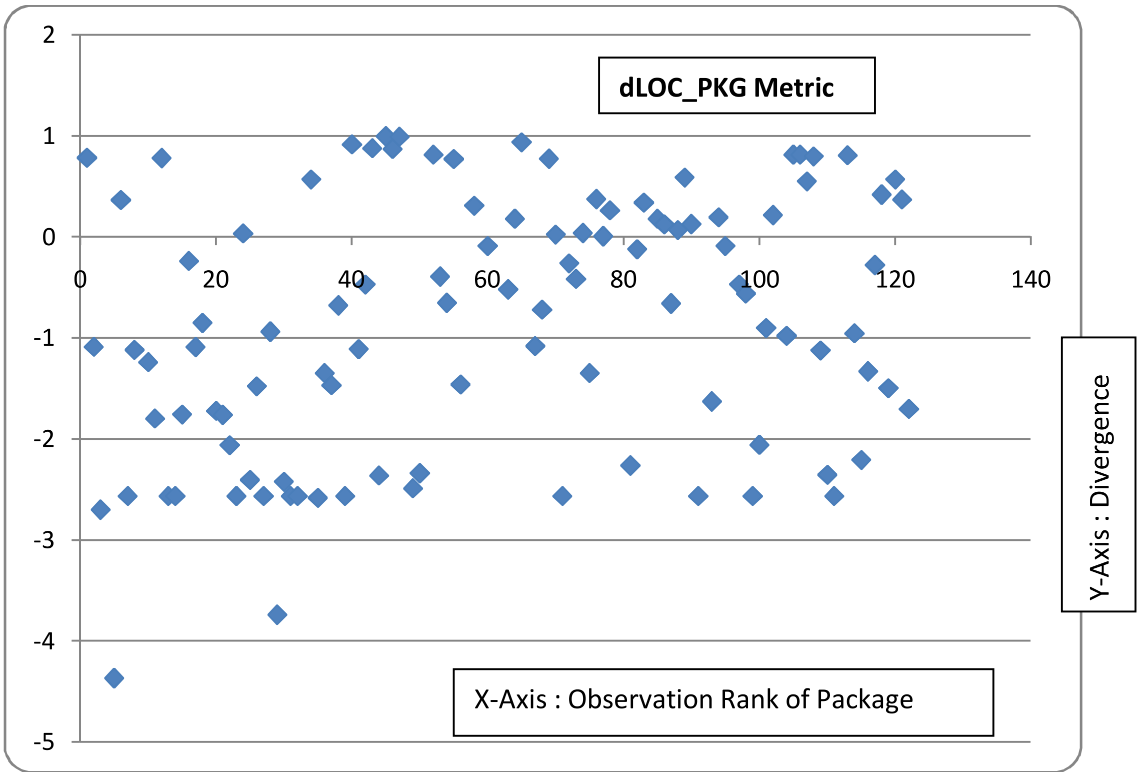


Figure 6 : Scatter Chart of Deviation – Test Metrics – Package Level - Apache Projects Only



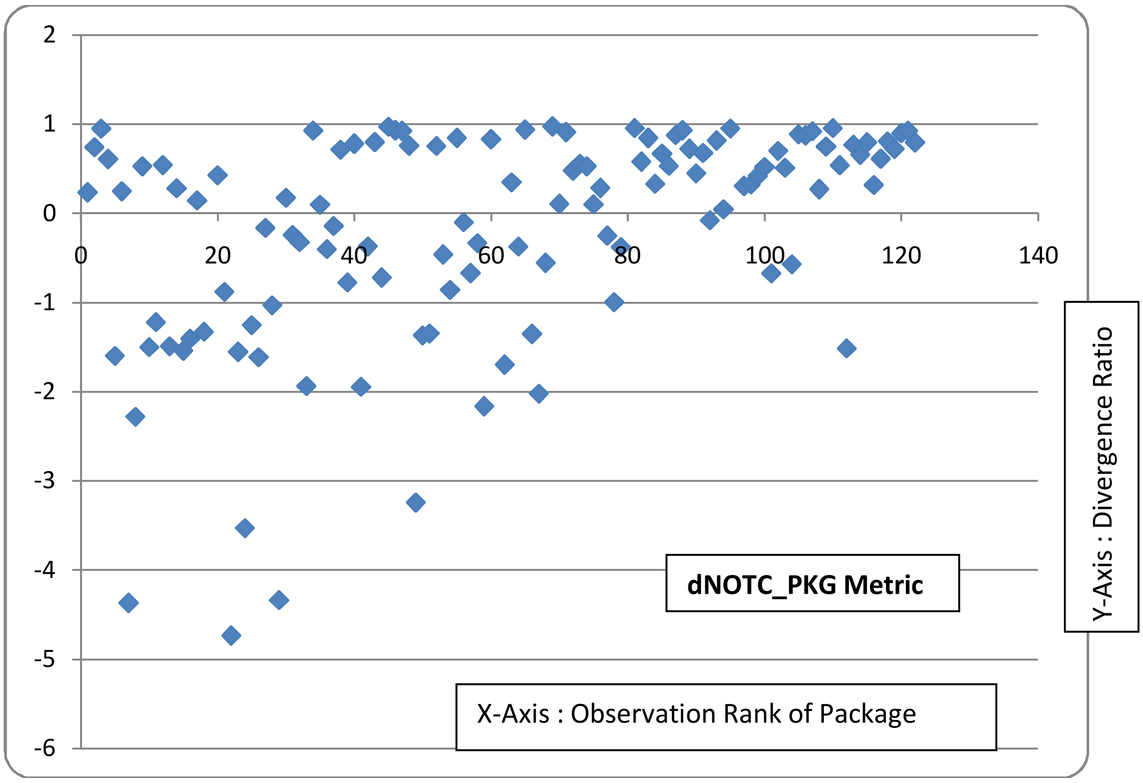


Figure 7 : Scatter Chart of Deviation – Test Metrics - Package Level - JBoss Projects Only

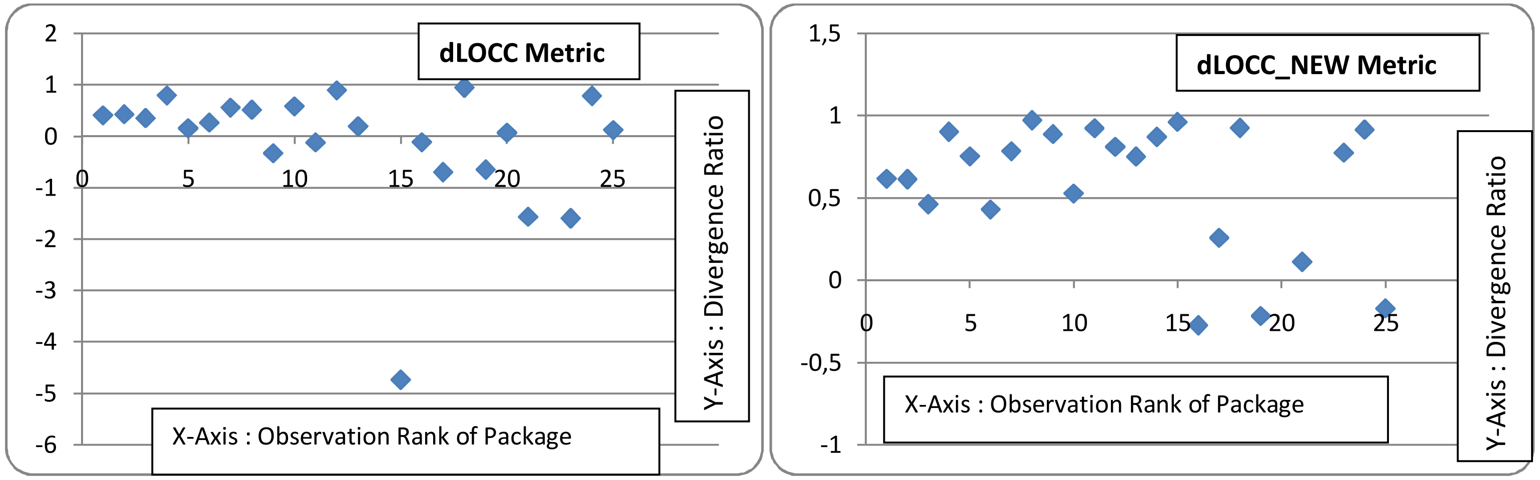


Figure 8 : Scatter Chart of Deviation – Test Metrics - Package Level - Apache Ant

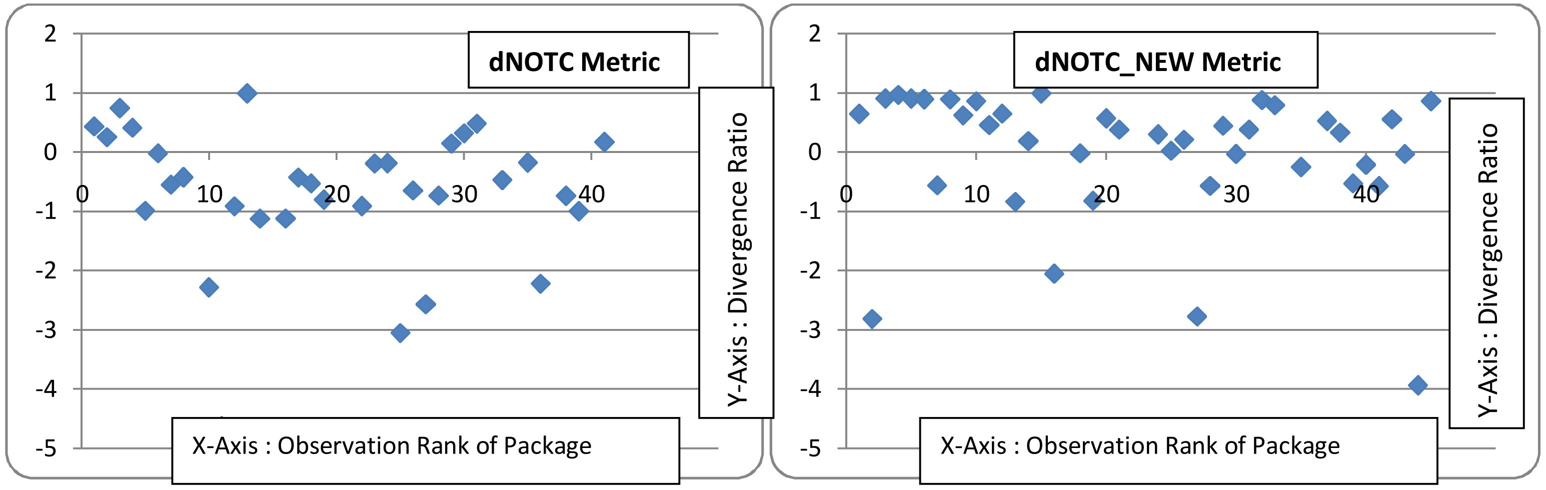


Figure 9 : Scatter Chart of Deviation – Test Metrics - Package Level - Apache Lucene

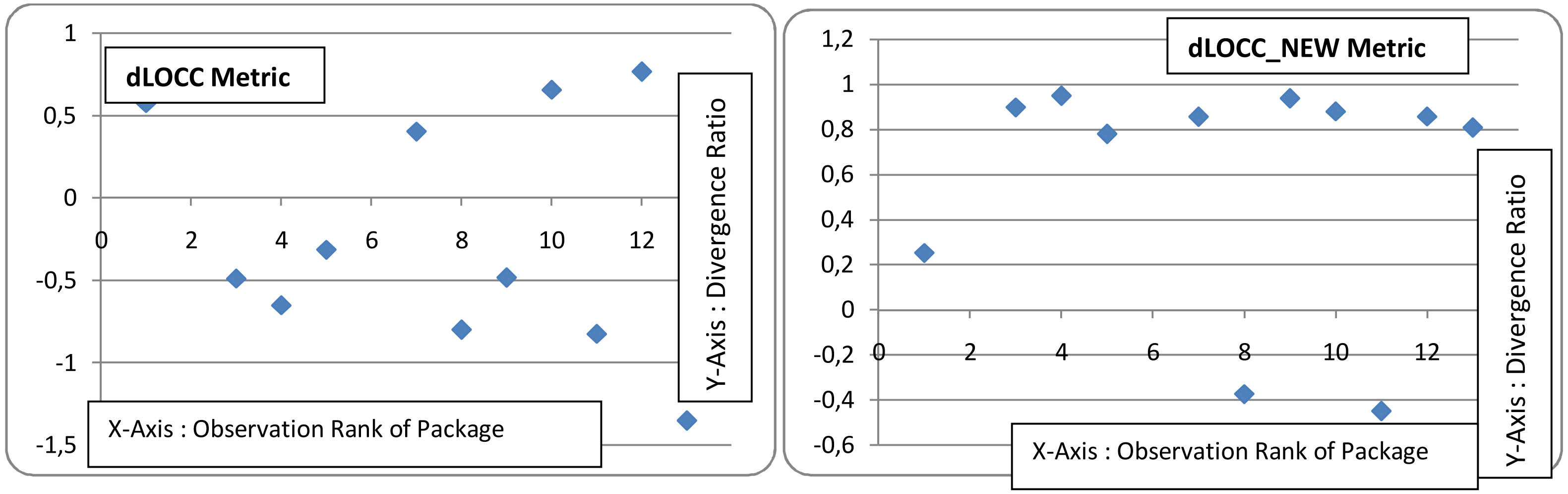


Figure 10 : Scatter Chart of Deviation – Test Metrics - Package Level - Apache Mina

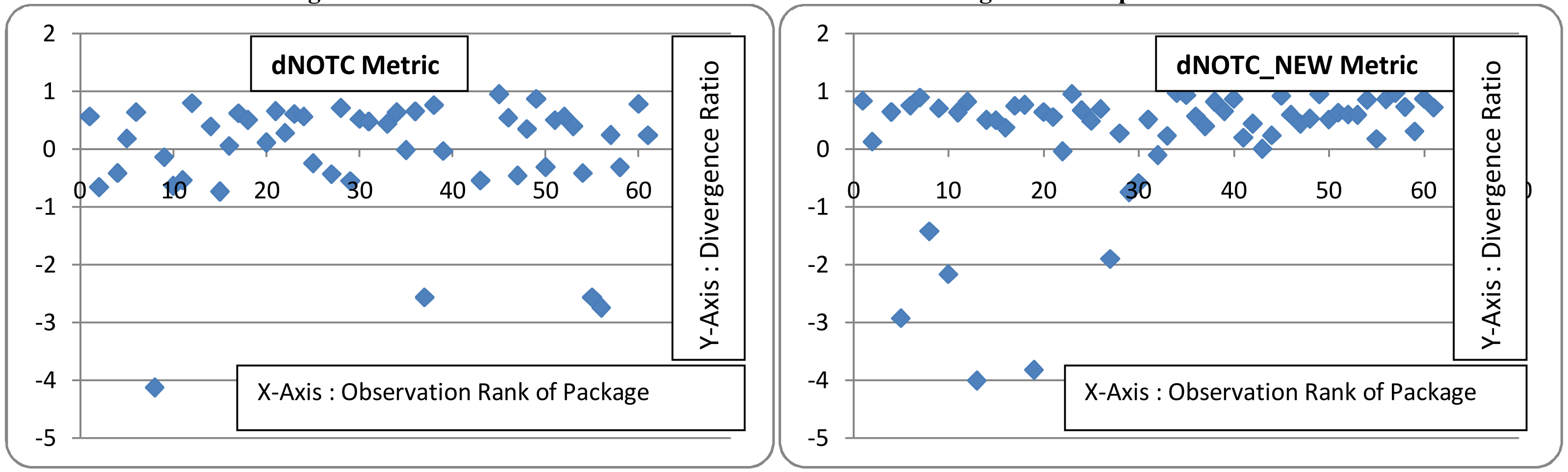


Figure 11 : Scatter Chart of Deviation – Test Metrics - Package Level - Apache Geronimo

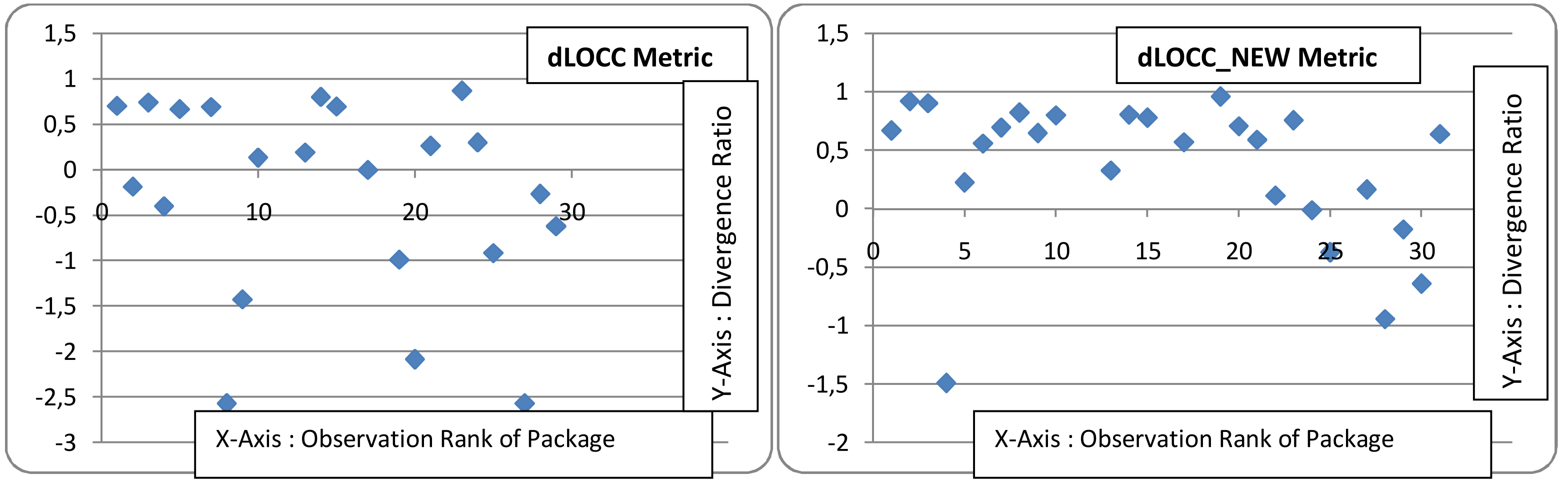


Figure 12 : Scatter Chart of Deviation – Test Metrics - Package Level - Apache Wicket

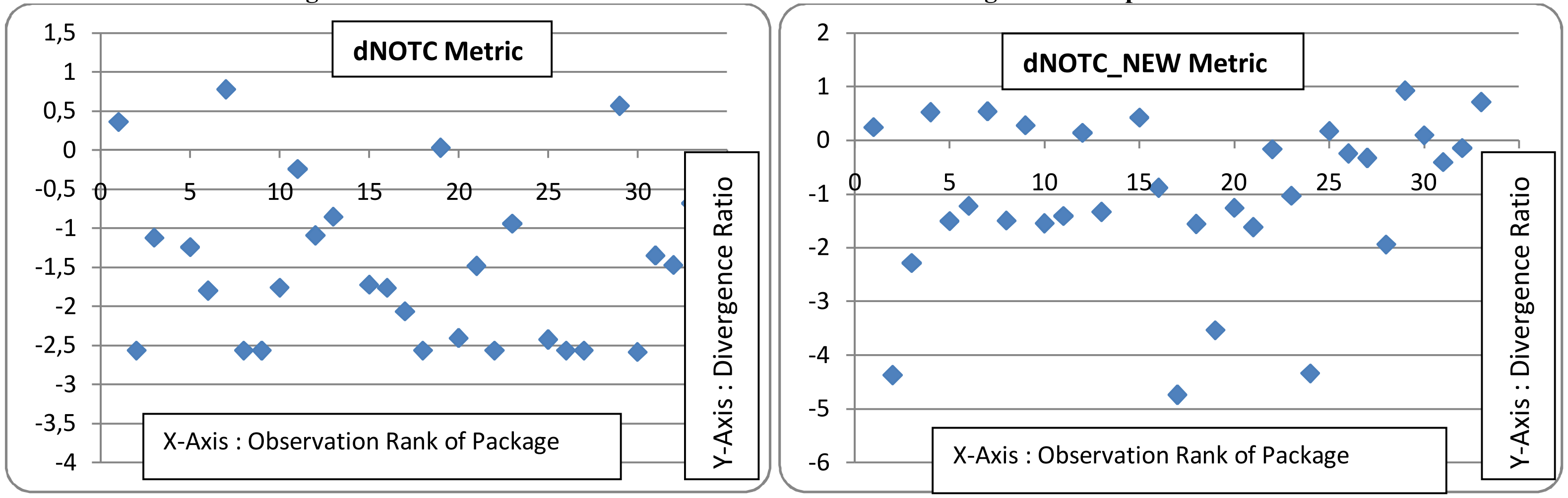


Figure 13 : Scatter Chart of Deviation – Test Metrics - Package Level - JBoss Cache

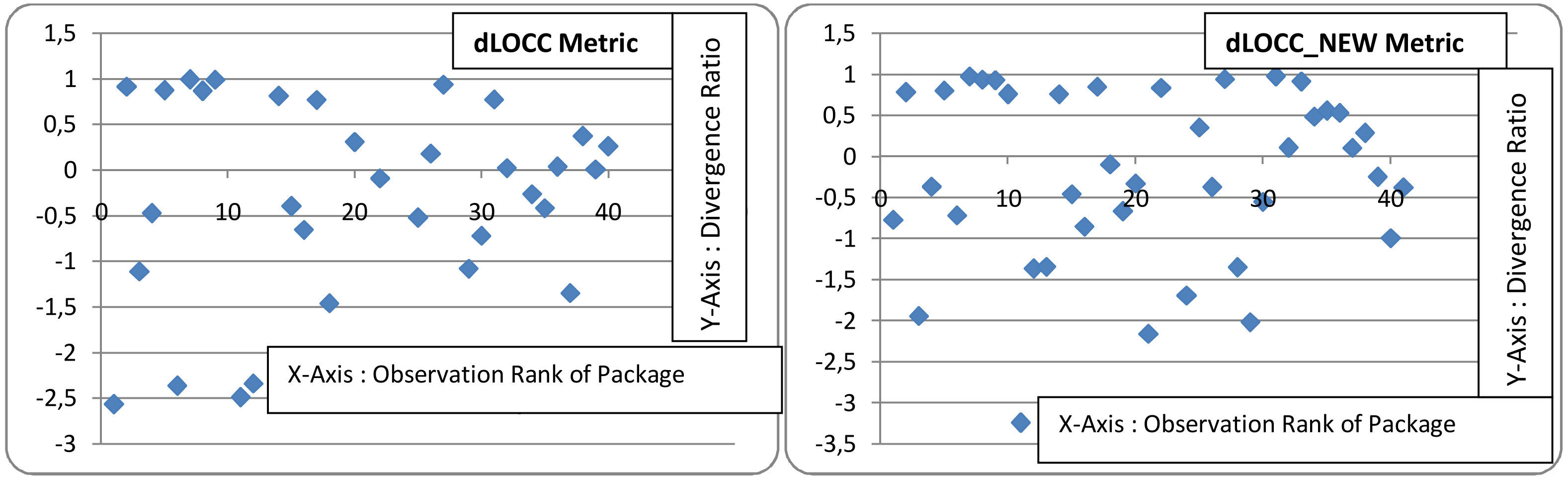


Figure 14 : Scatter Chart of Deviation – Test Metrics - Package Level - JBoss Drools

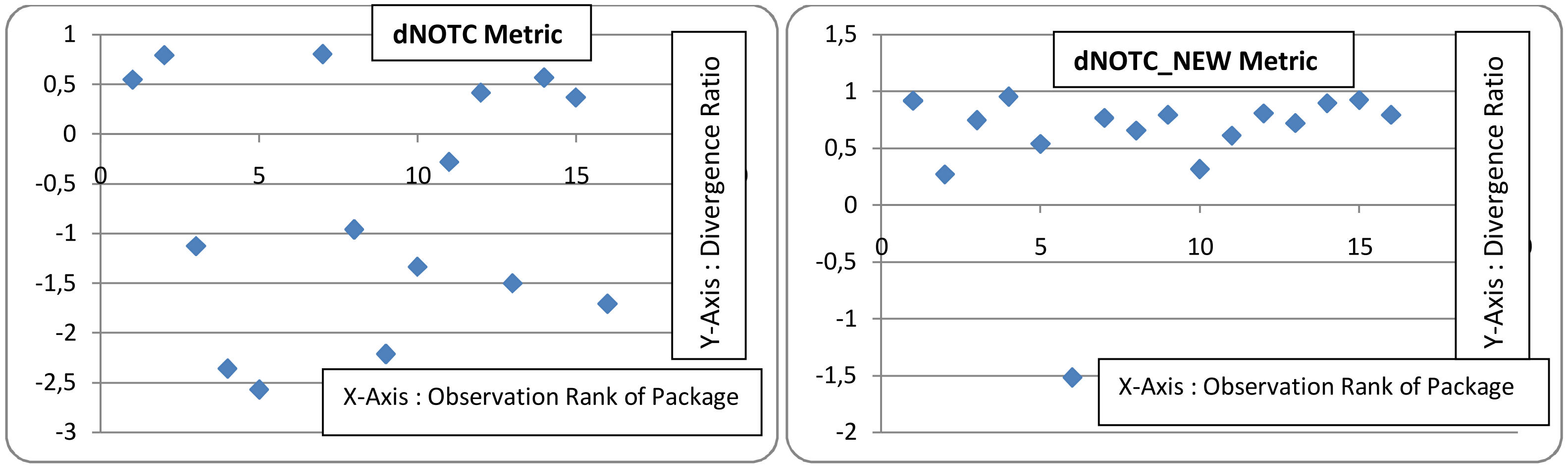


Figure 15 : Scatter Chart of Deviation – Test Metrics - Package Level - JBoss Richfaces

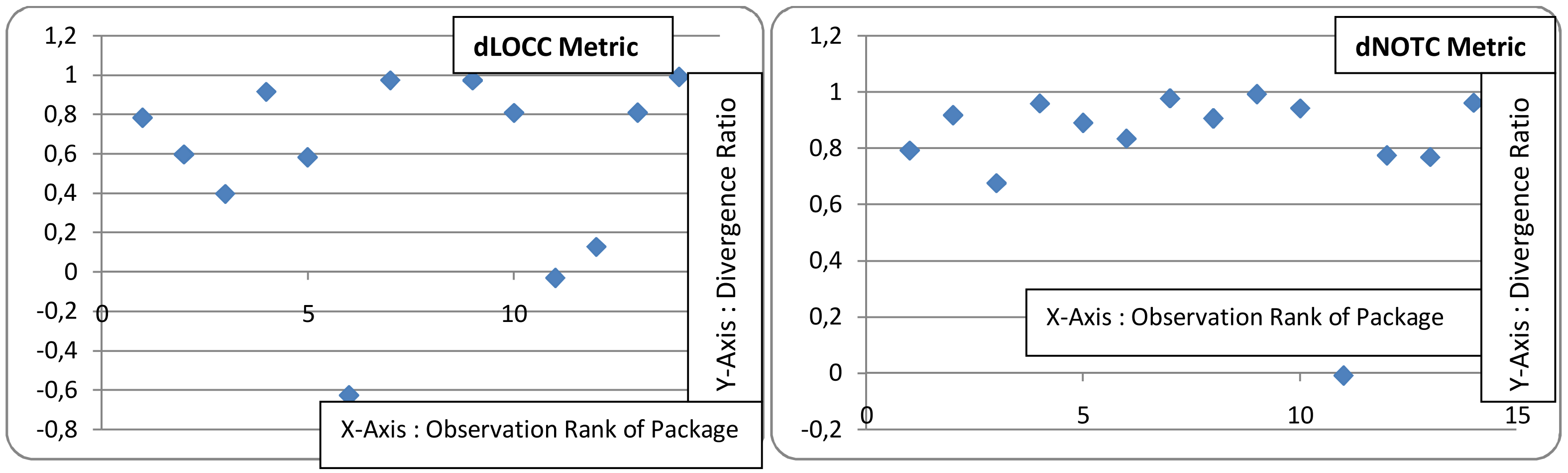


Figure 16 : Scatter Chart of Deviation – Test Metrics - Package Level - Apache JackRabbit

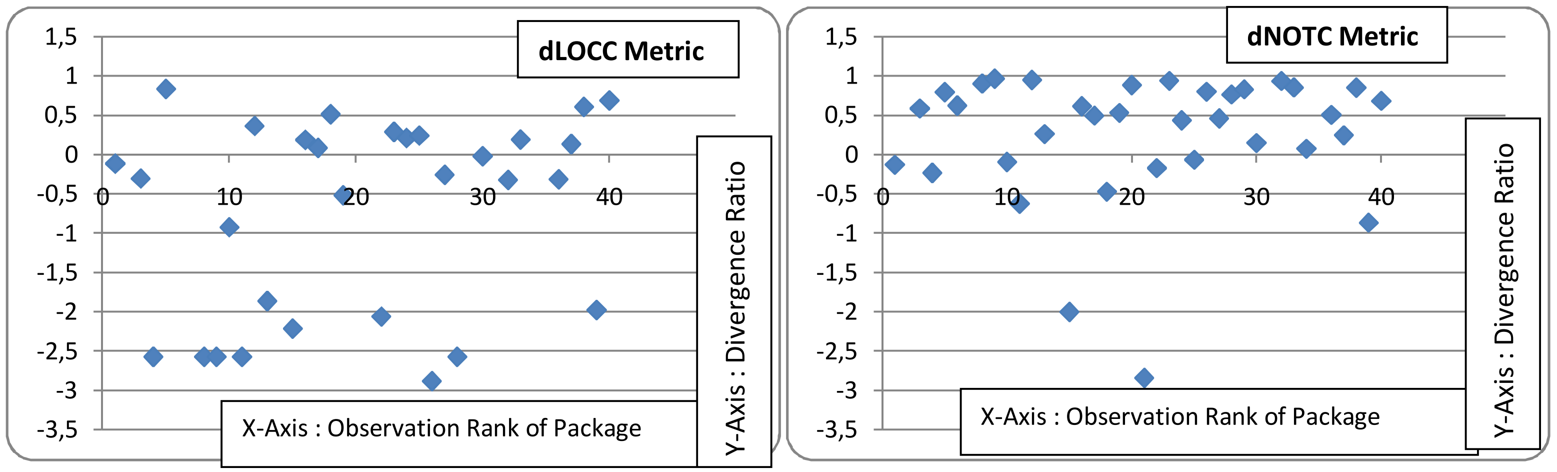


Figure 17 : Scatter Chart of Deviation – Test Metrics - Package Level - Apache ActiveMQ

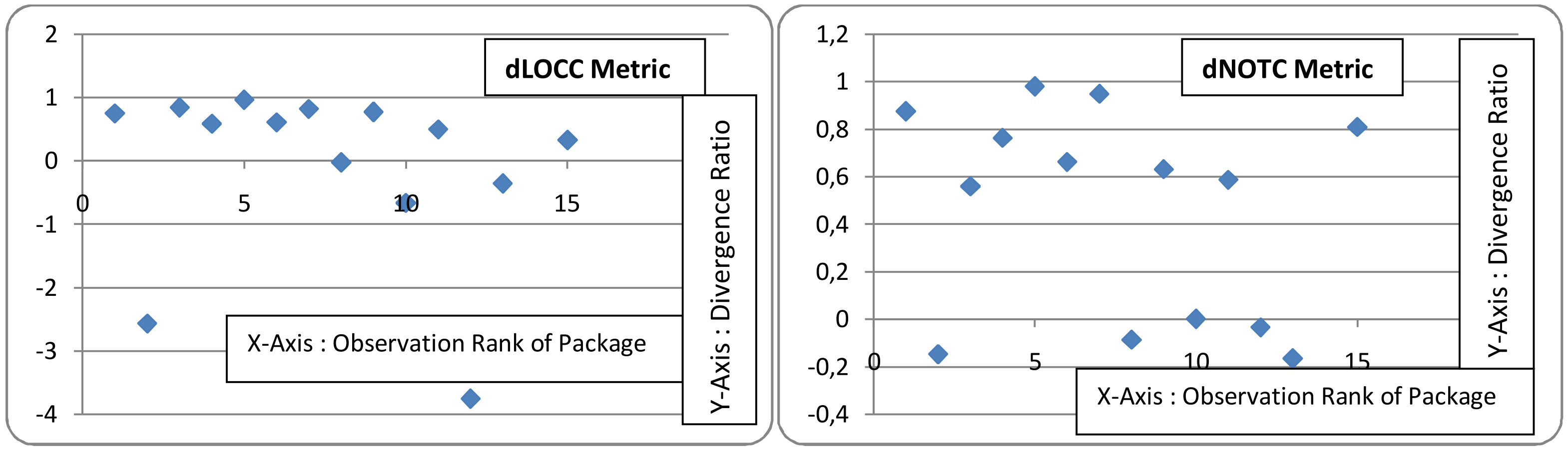


Figure 18 : Scatter Chart of Deviation – Test Metrics - Package Level - Apache ODE

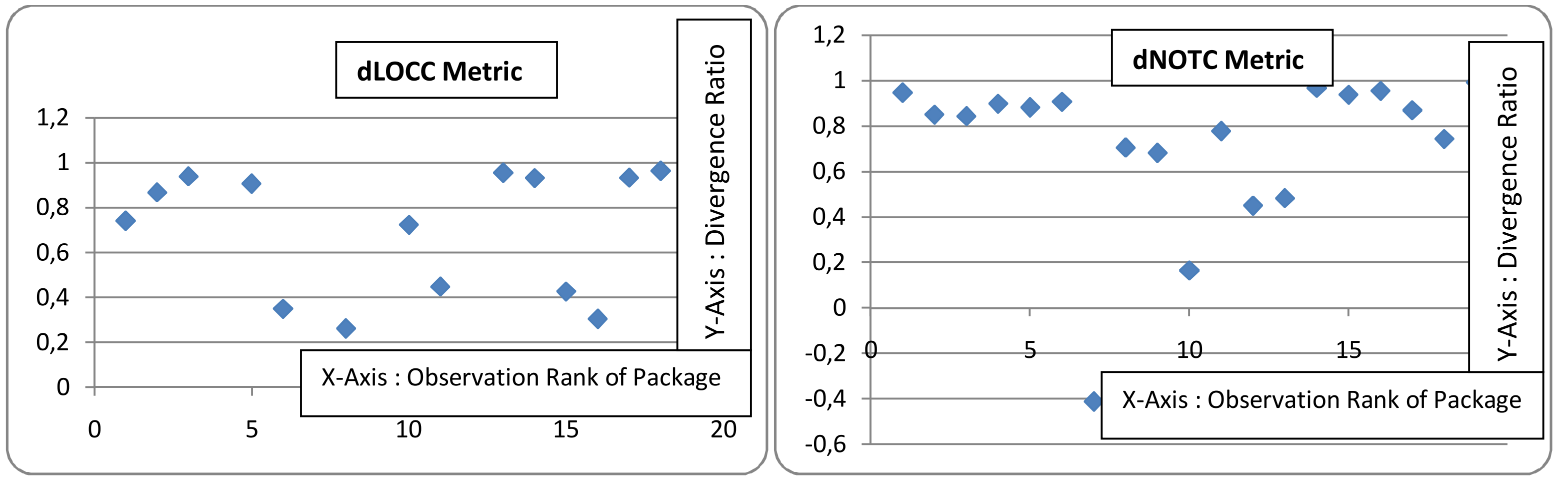


Figure 19 : Scatter Chart of Deviation – Test Metrics - Package Level - Apache OpenEJB

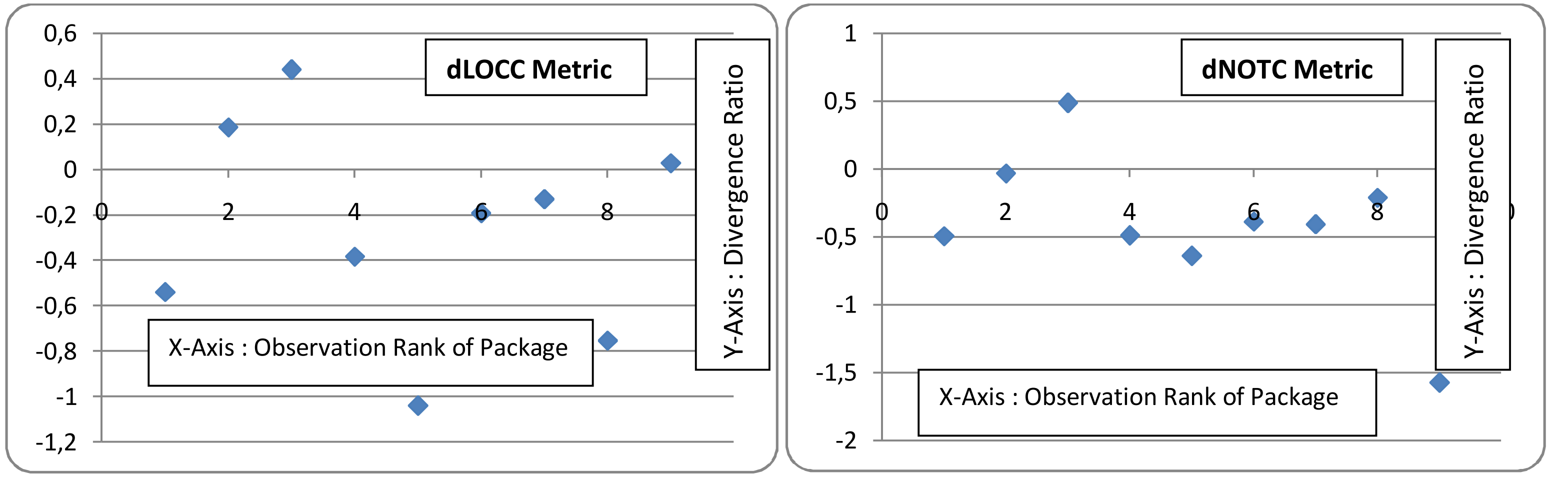


Figure 20 : Scatter Chart of Deviation – Test Metrics - Package Level - Apache OJB

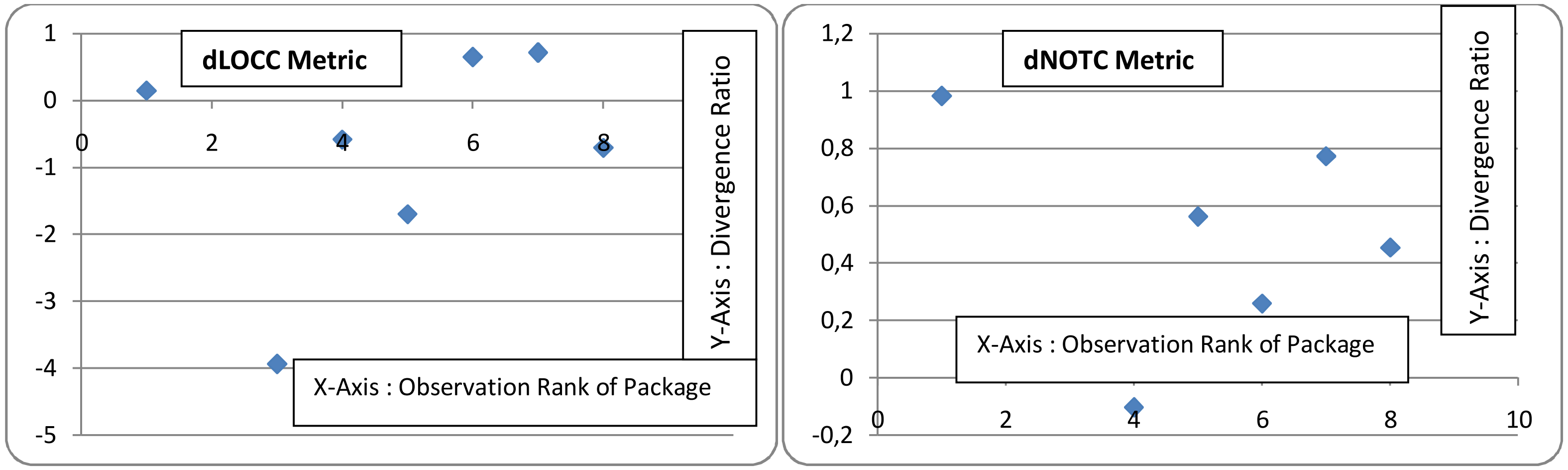


Figure 21 : Scatter Chart of Deviation – Test Metrics - Package Level - Apache Struts

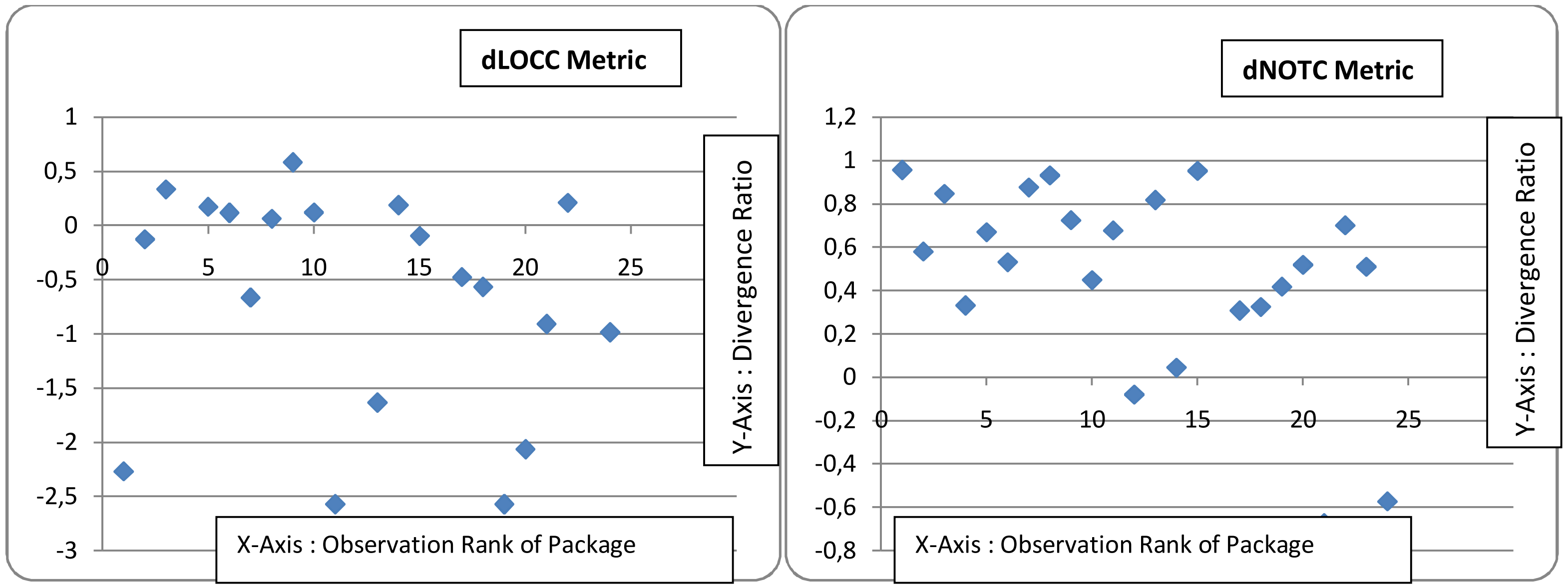


Figure 22 : Scatter Chart of Deviation – Test Metrics - Package Level - JBoss ESB

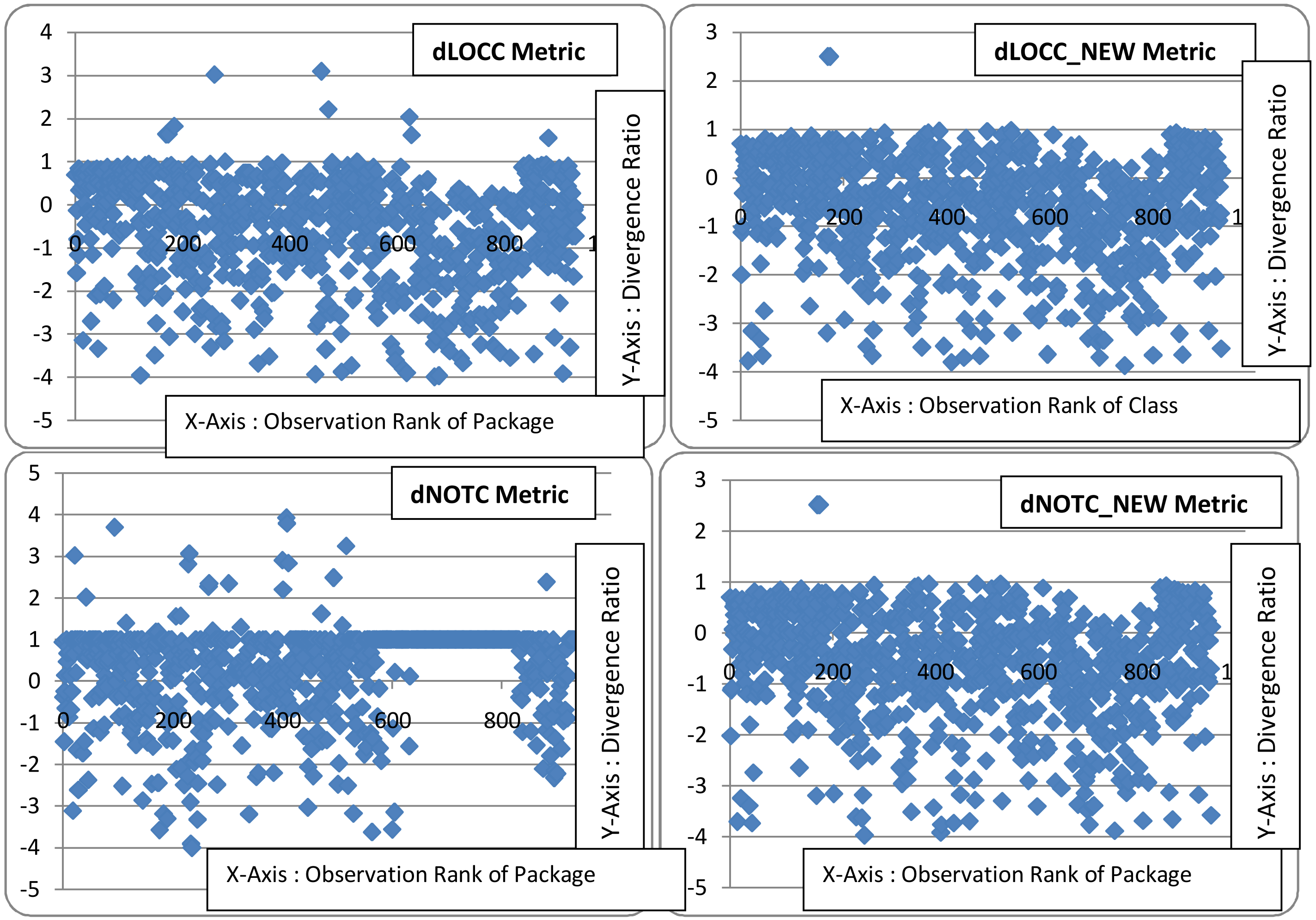


Figure 23 : Scatter Chart of Deviation – Test Metrics - Class Level - Apache Projects Only

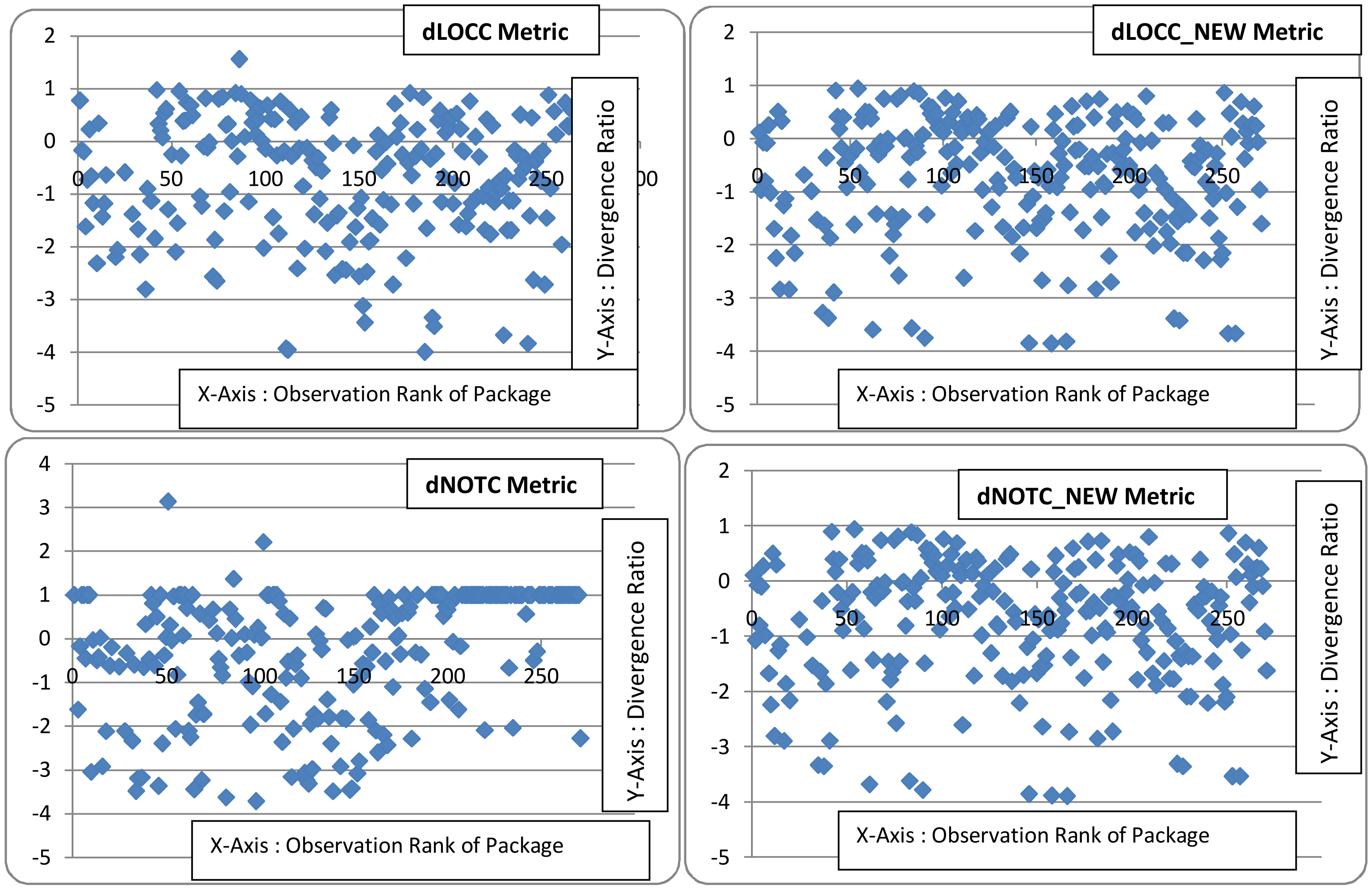


Figure 24 : Scatter Chart of Deviation – Test Metrics - Class Level - JBoss Projects Only

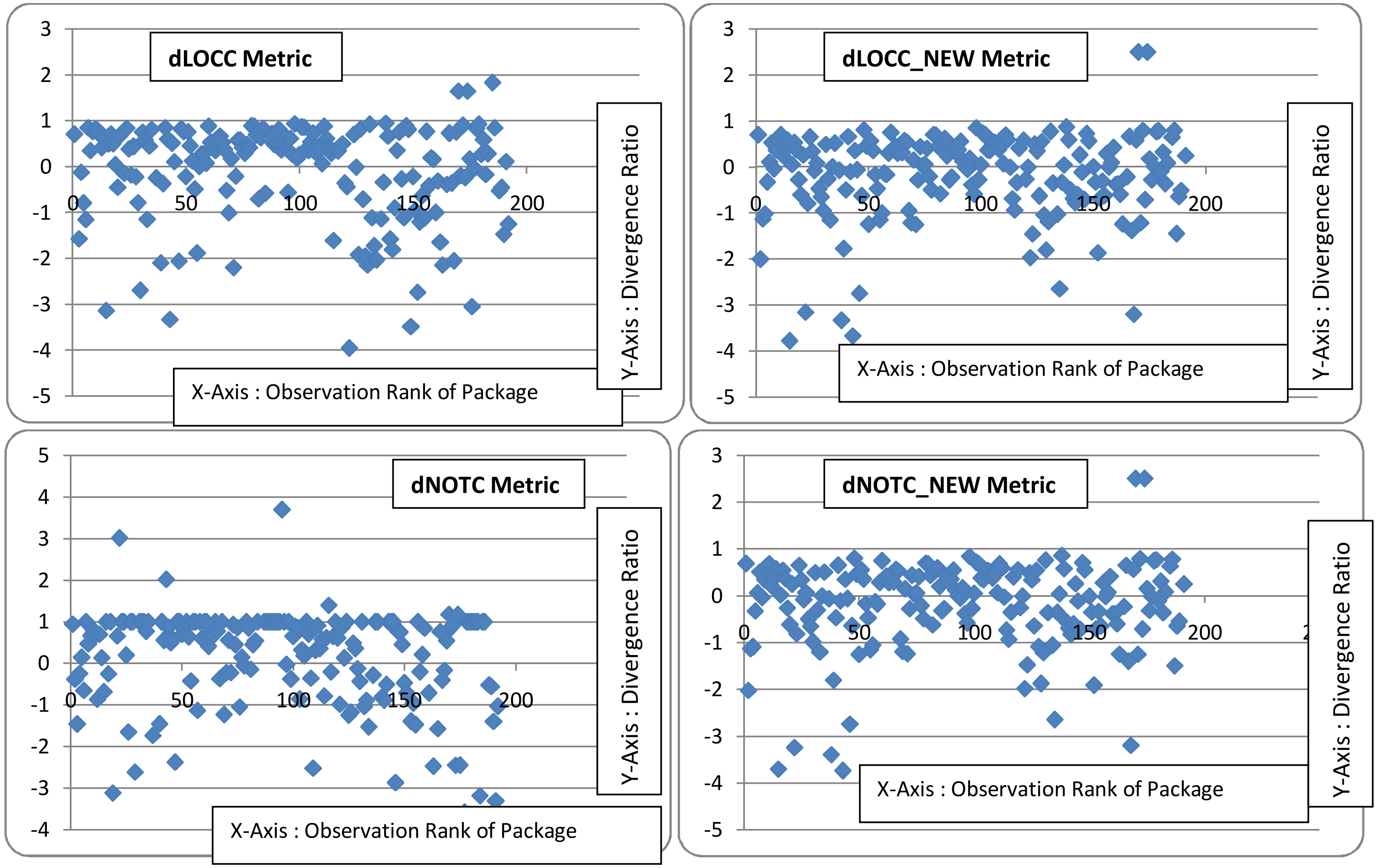


Figure 25 : Scatter Chart of Deviation – Test Metrics - Class Level – Apache Ant

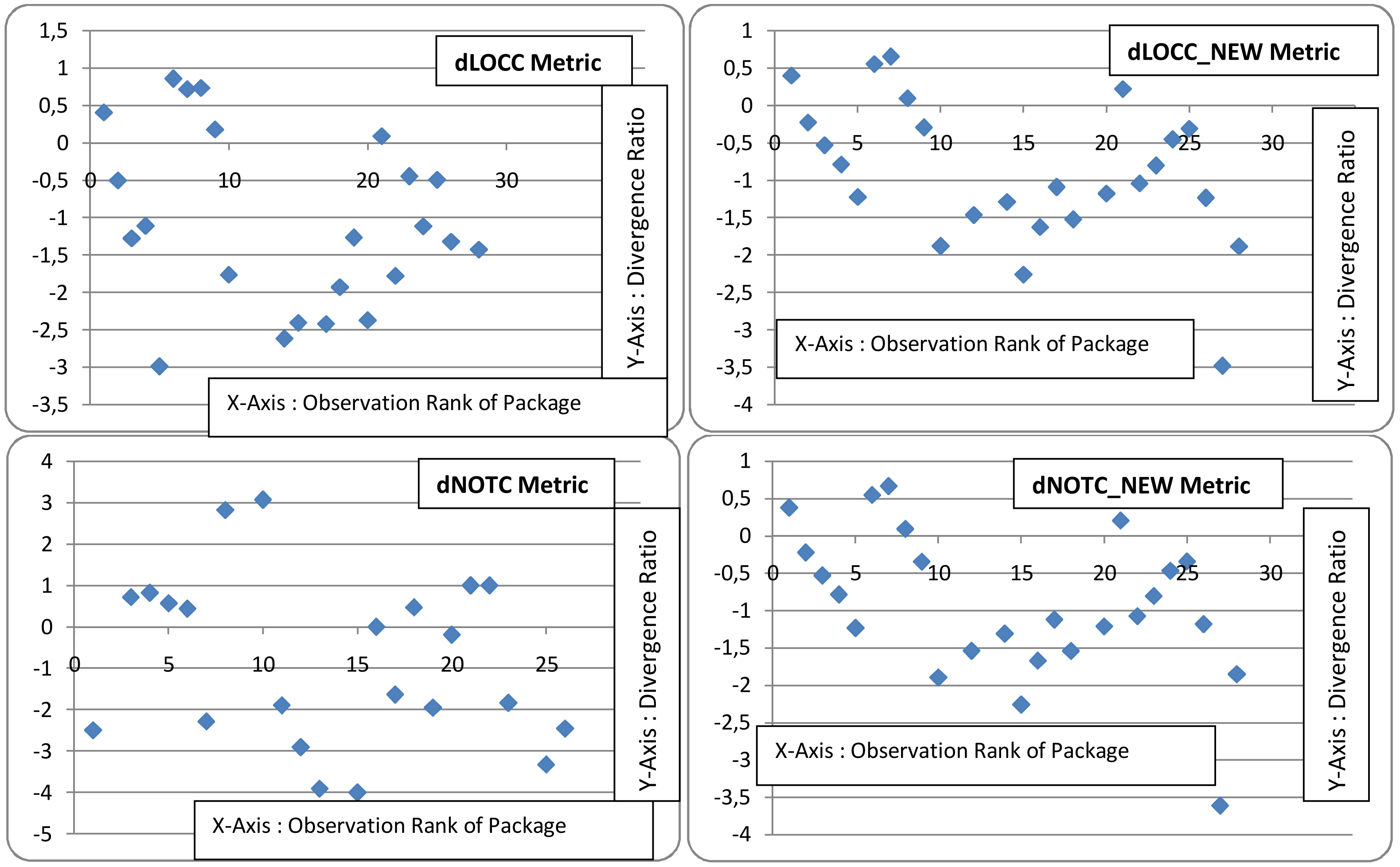


Figure 26 : Scatter Chart of Deviation – Test Metrics - Class Level – Apache Lucene

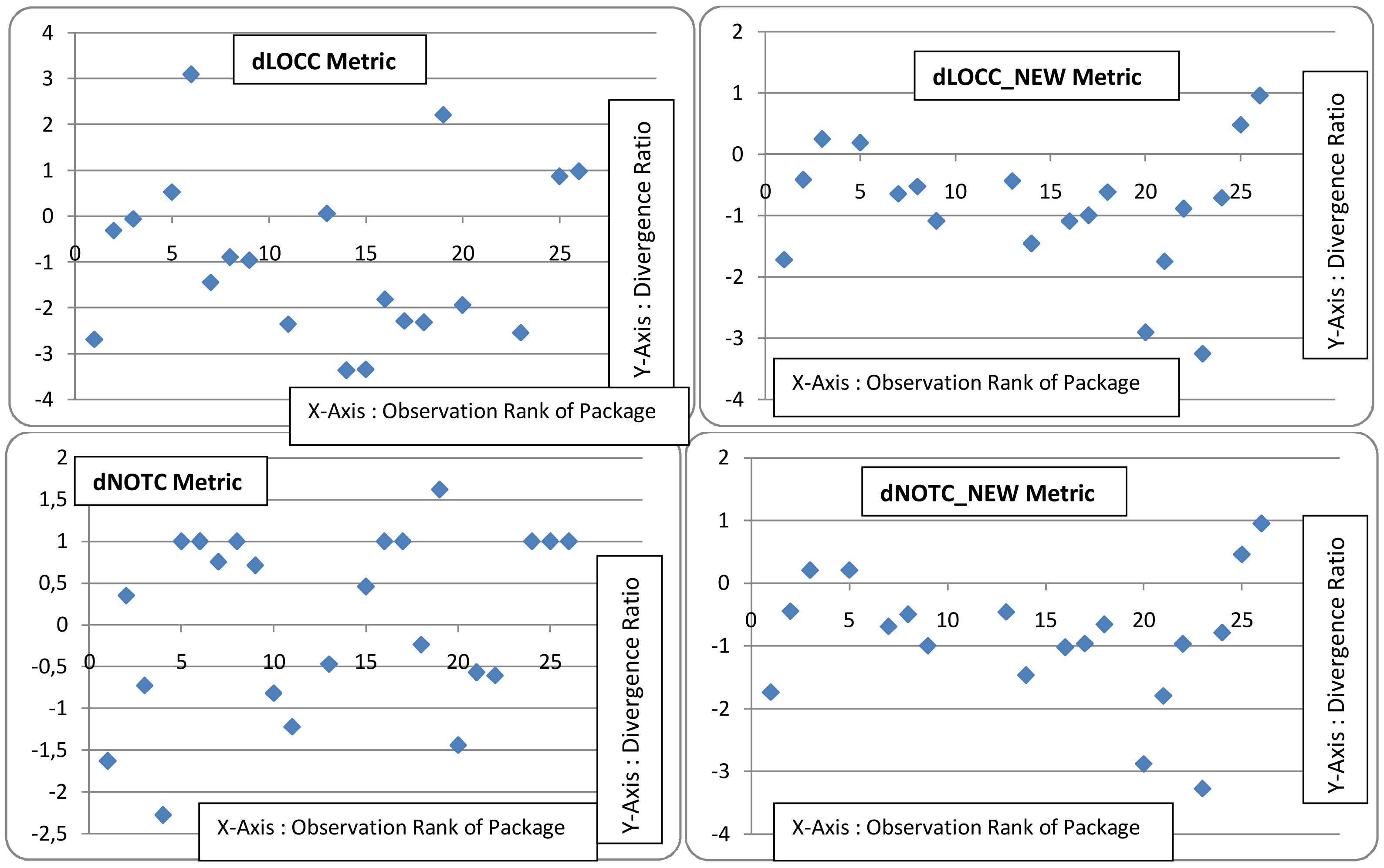


Figure 27 : Scatter Chart of Deviation – Test Metrics - Class Level – Apache Mina

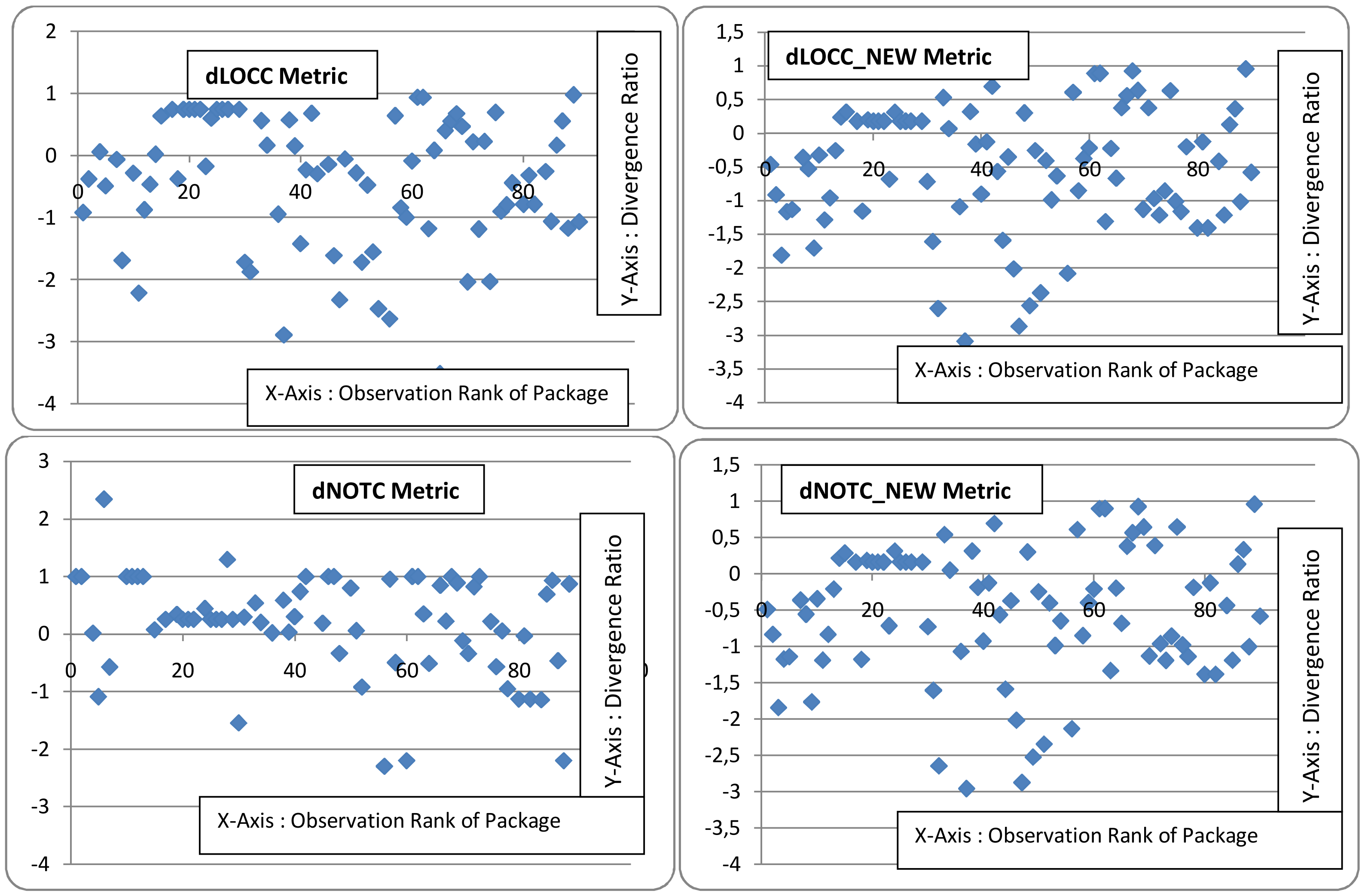


Figure 28 : Scatter Chart of Deviation – Test Metrics - Class Level – Apache Geronimo

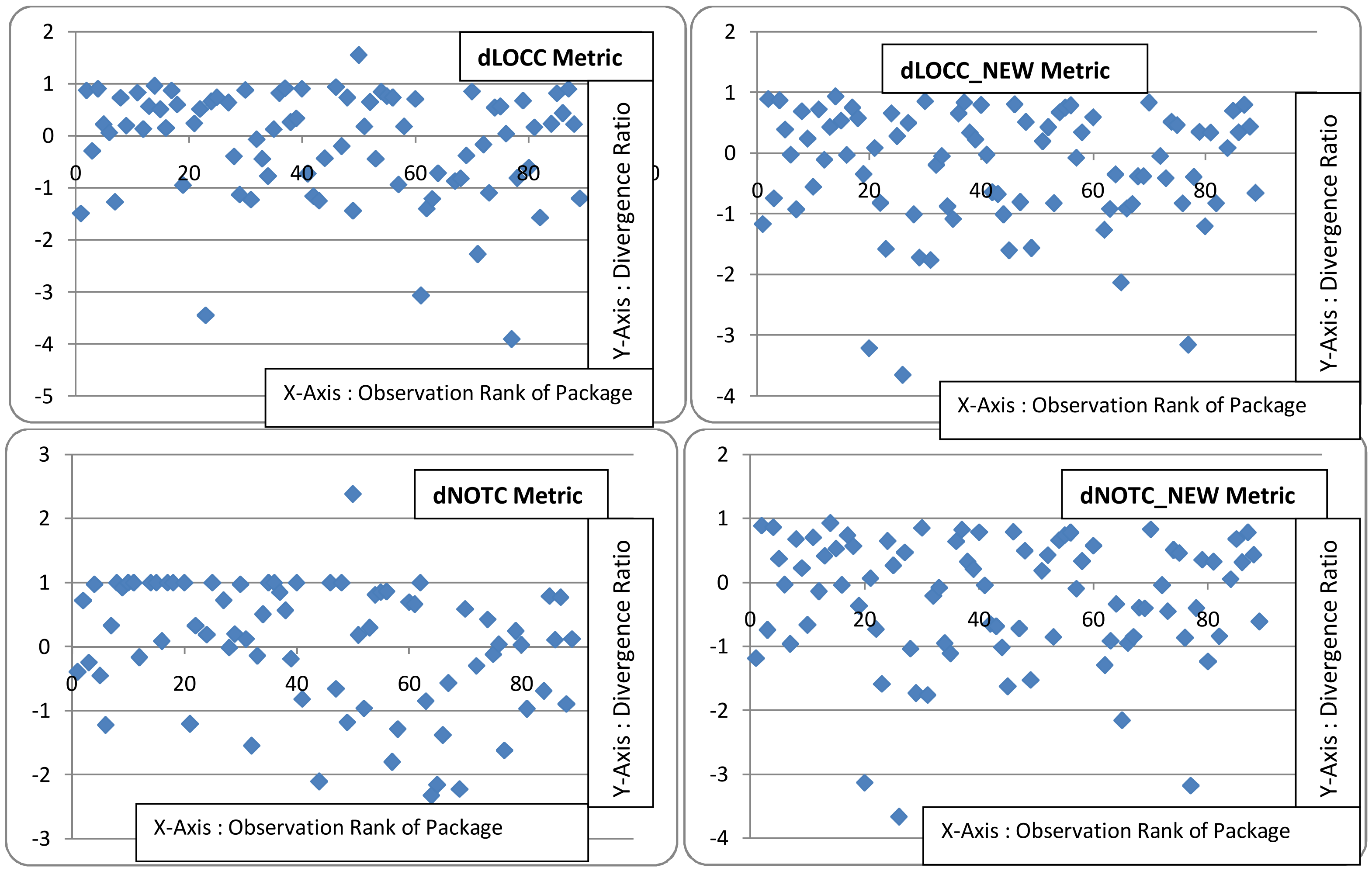


Figure 29 : Scatter Chart of Deviation – Test Metrics - Class Level – Apache Wicket

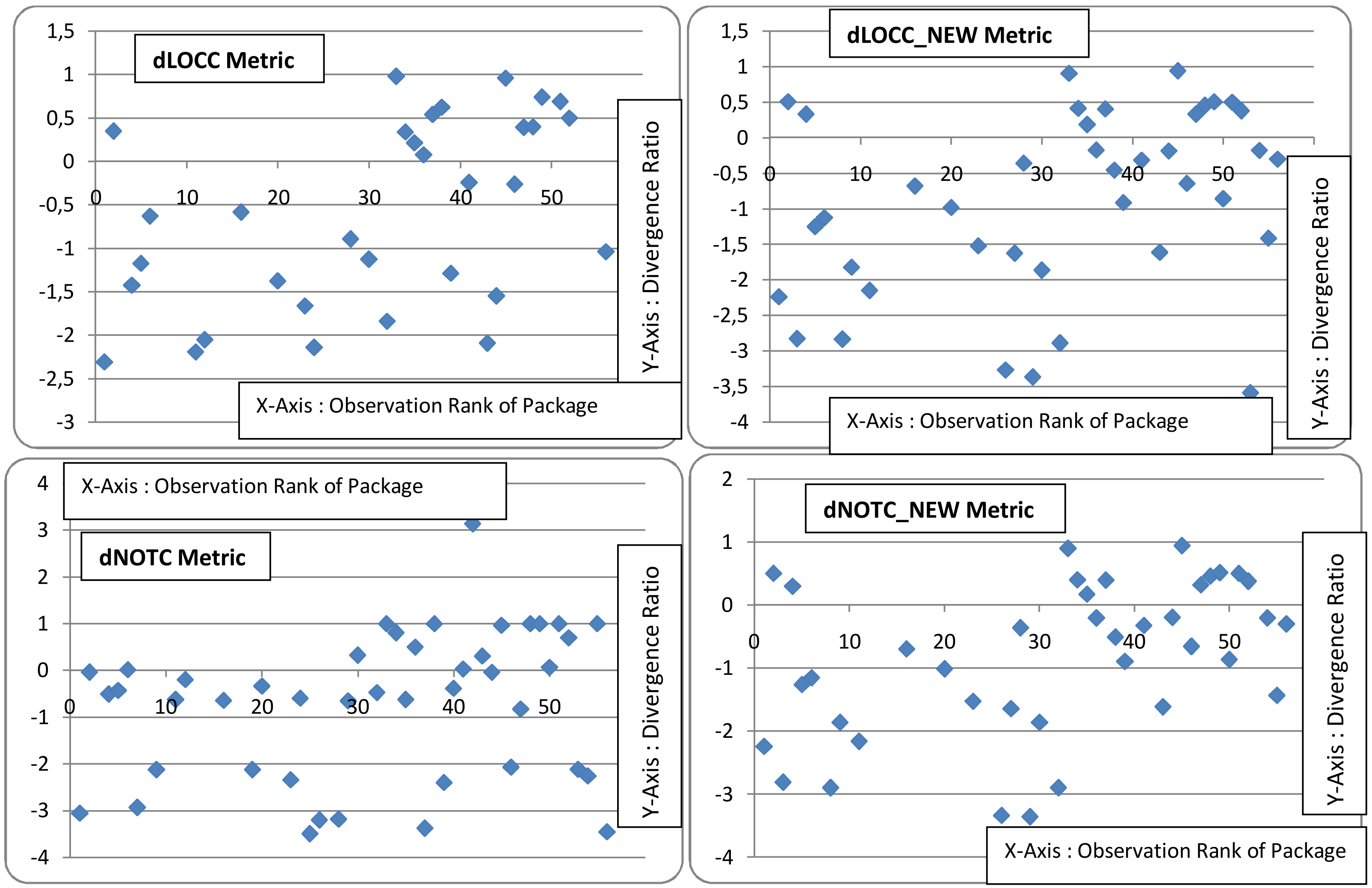


Figure 30 : Scatter Chart of Deviation – Test Metrics - Class Level – JBoss Cache

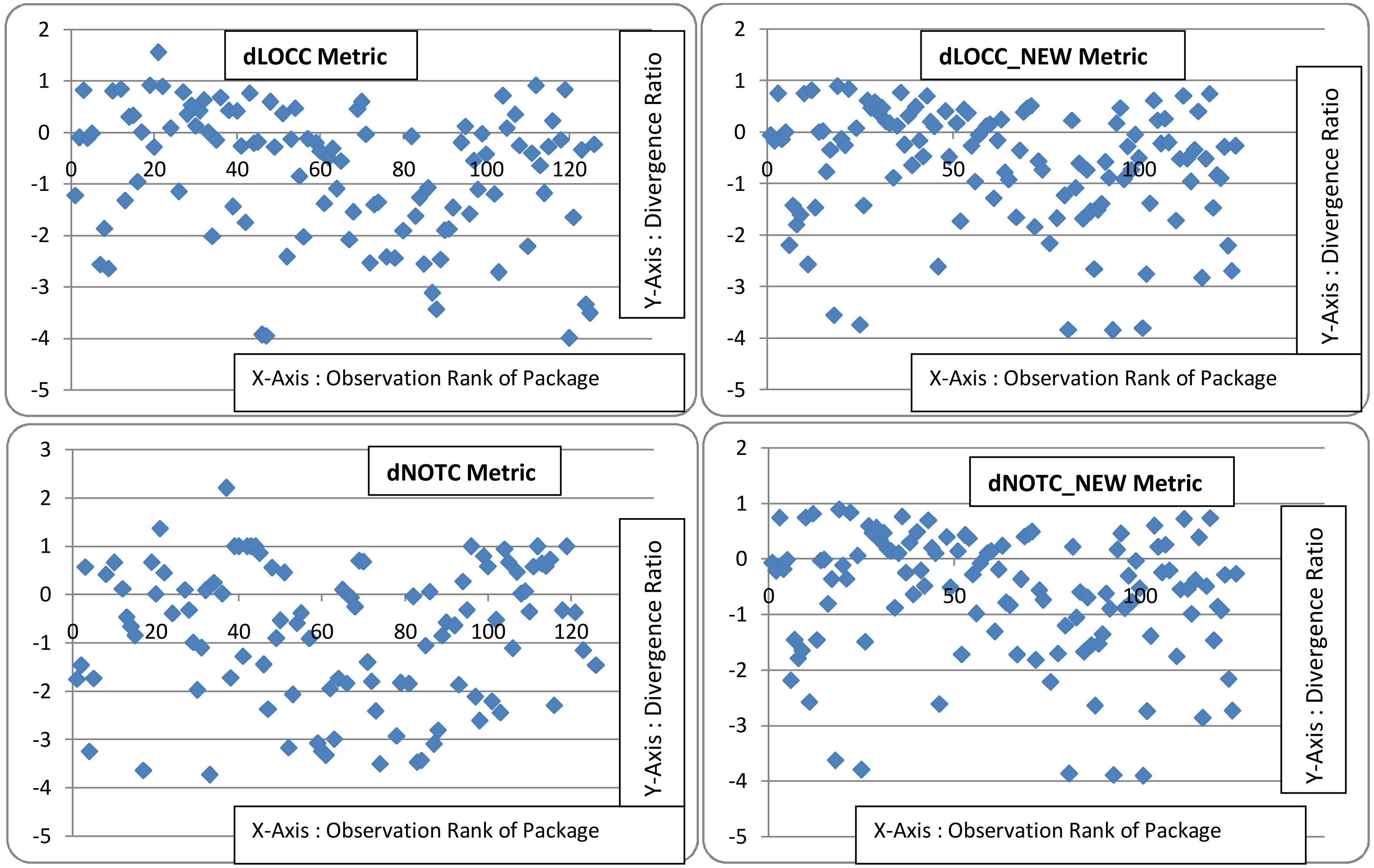


Figure 31 : Scatter Chart of Deviation – Test Metrics - Class Level – JBoss Drools

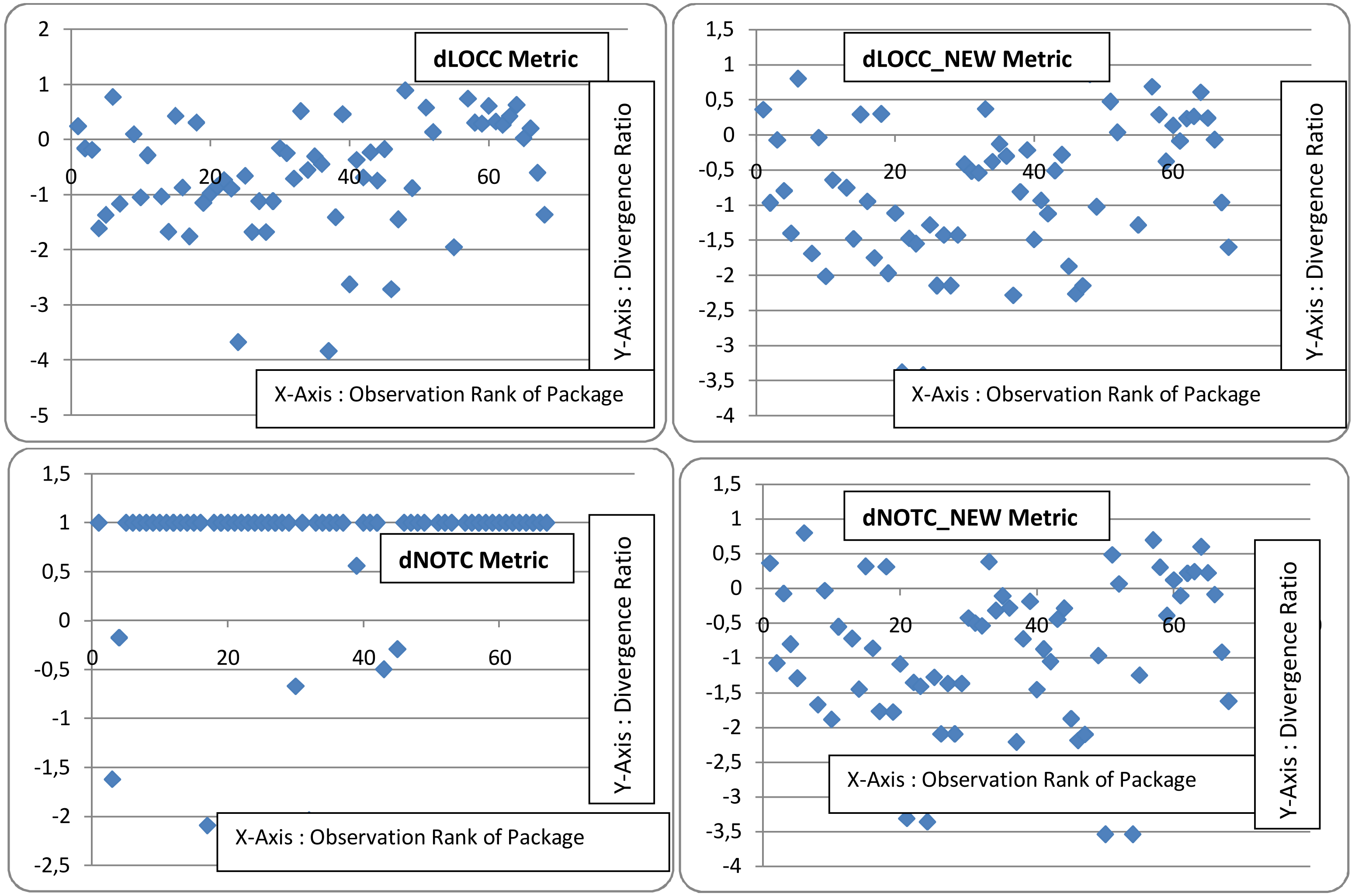


Figure 32 : Scatter Chart of Deviation – Test Metrics - Class Level – JBoss Richfaces

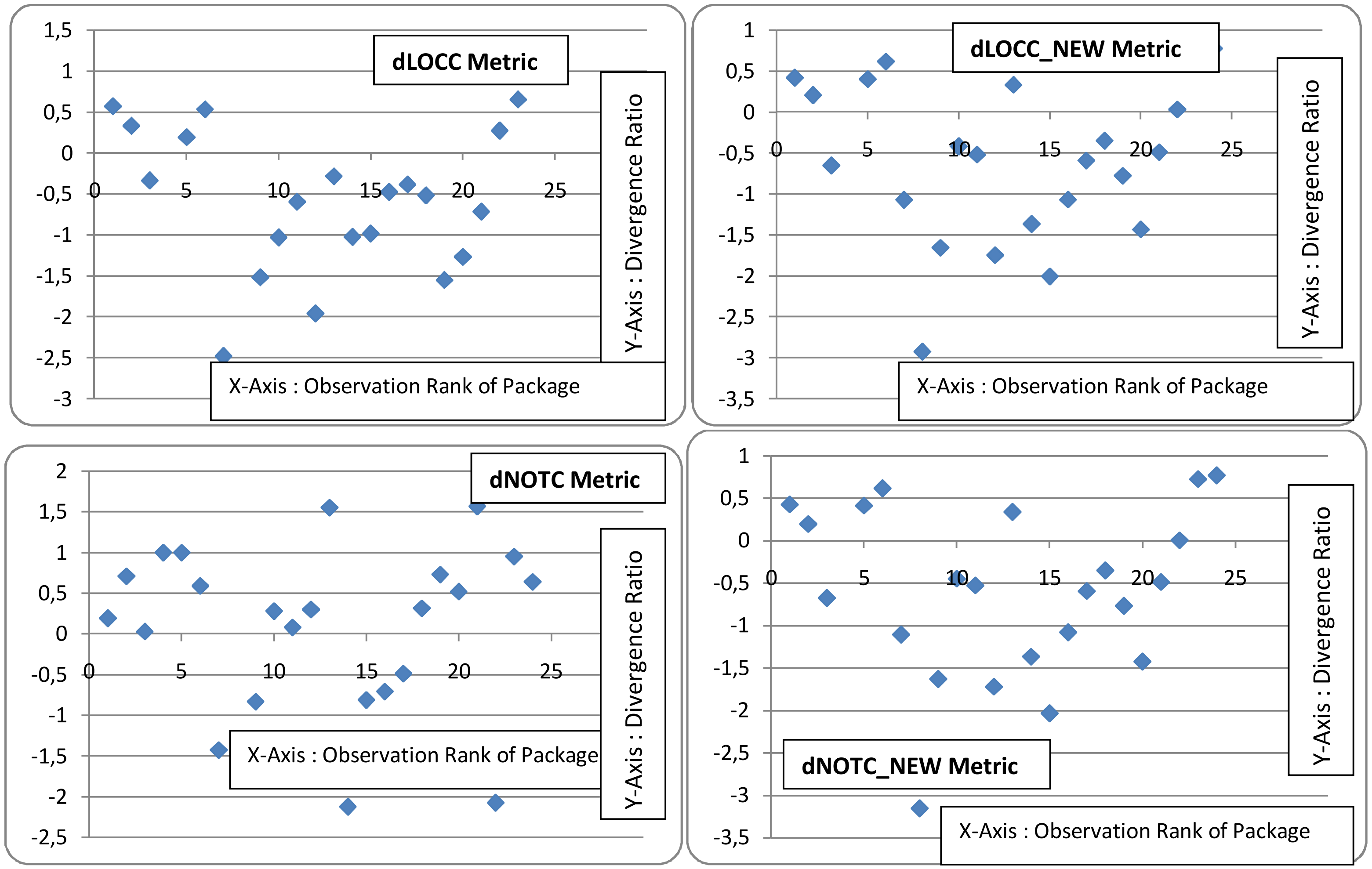


Figure 33 : Scatter Chart of Deviation – Test Metrics - Class Level – Apache JackRabbit

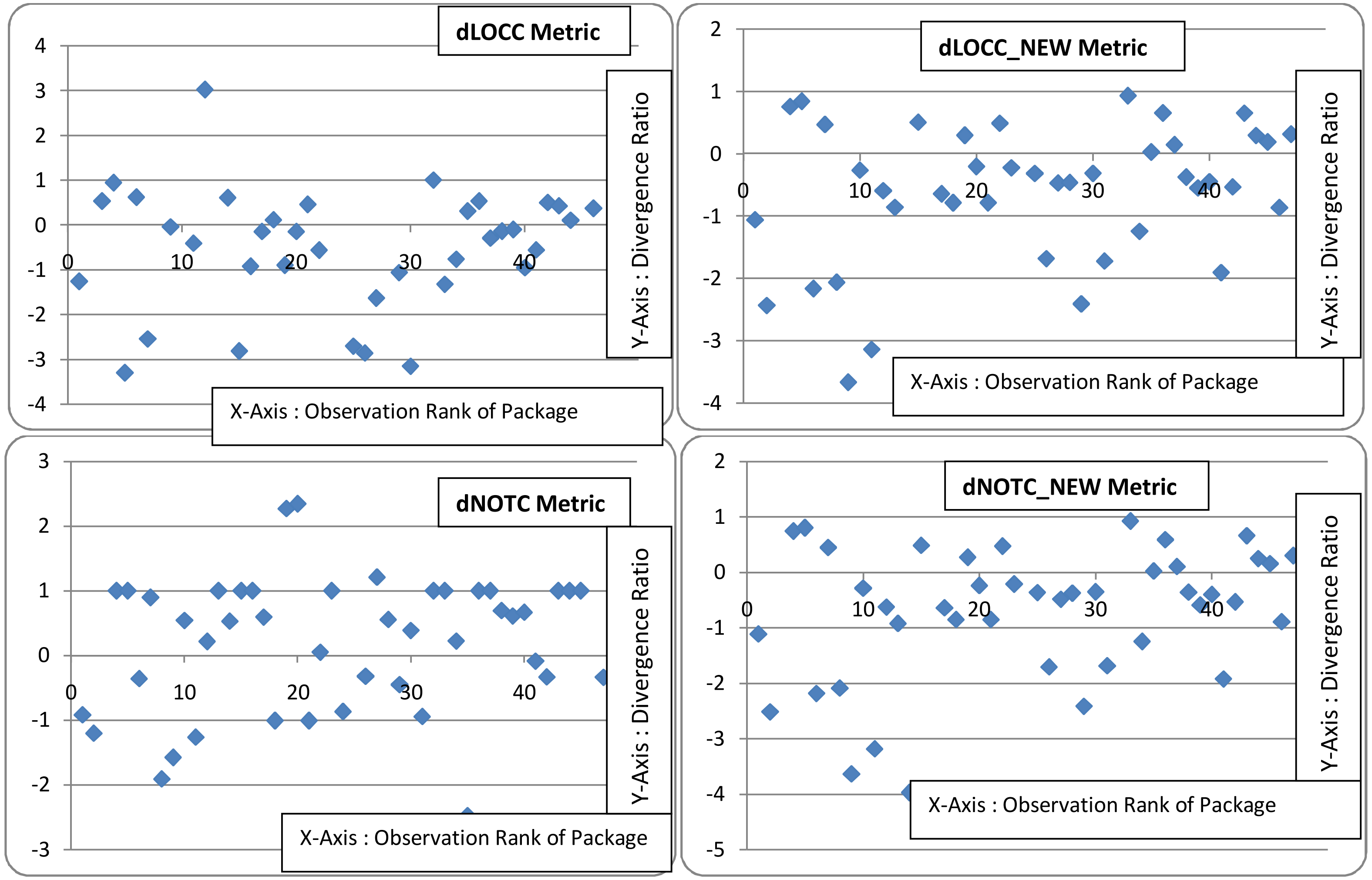


Figure 34 : Scatter Chart of Deviation – Test Metrics - Class Level – Apache ActiveMQ

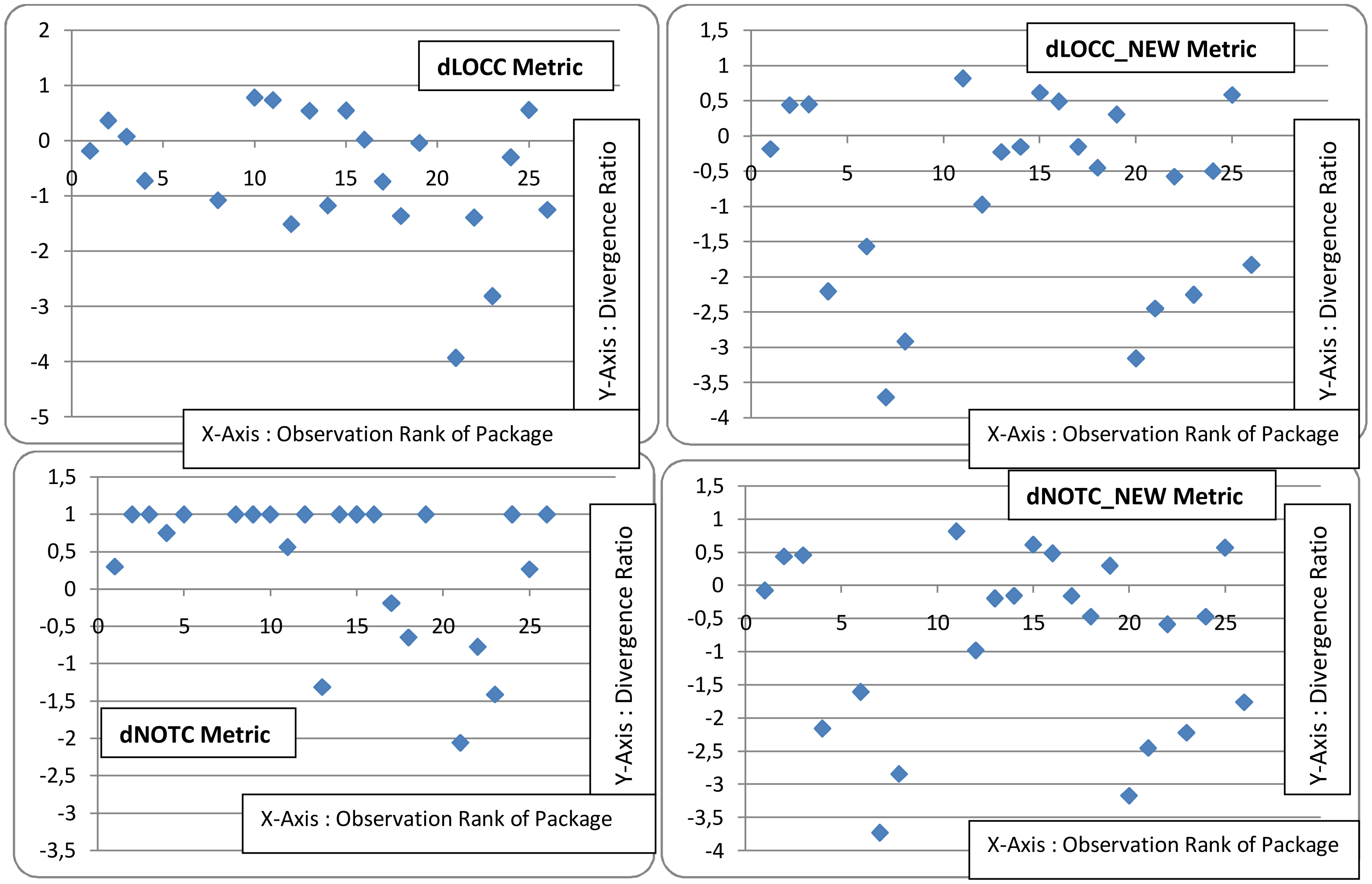


Figure 35 : Scatter Chart of Deviation – Test Metrics - Class Level – Apache Maven

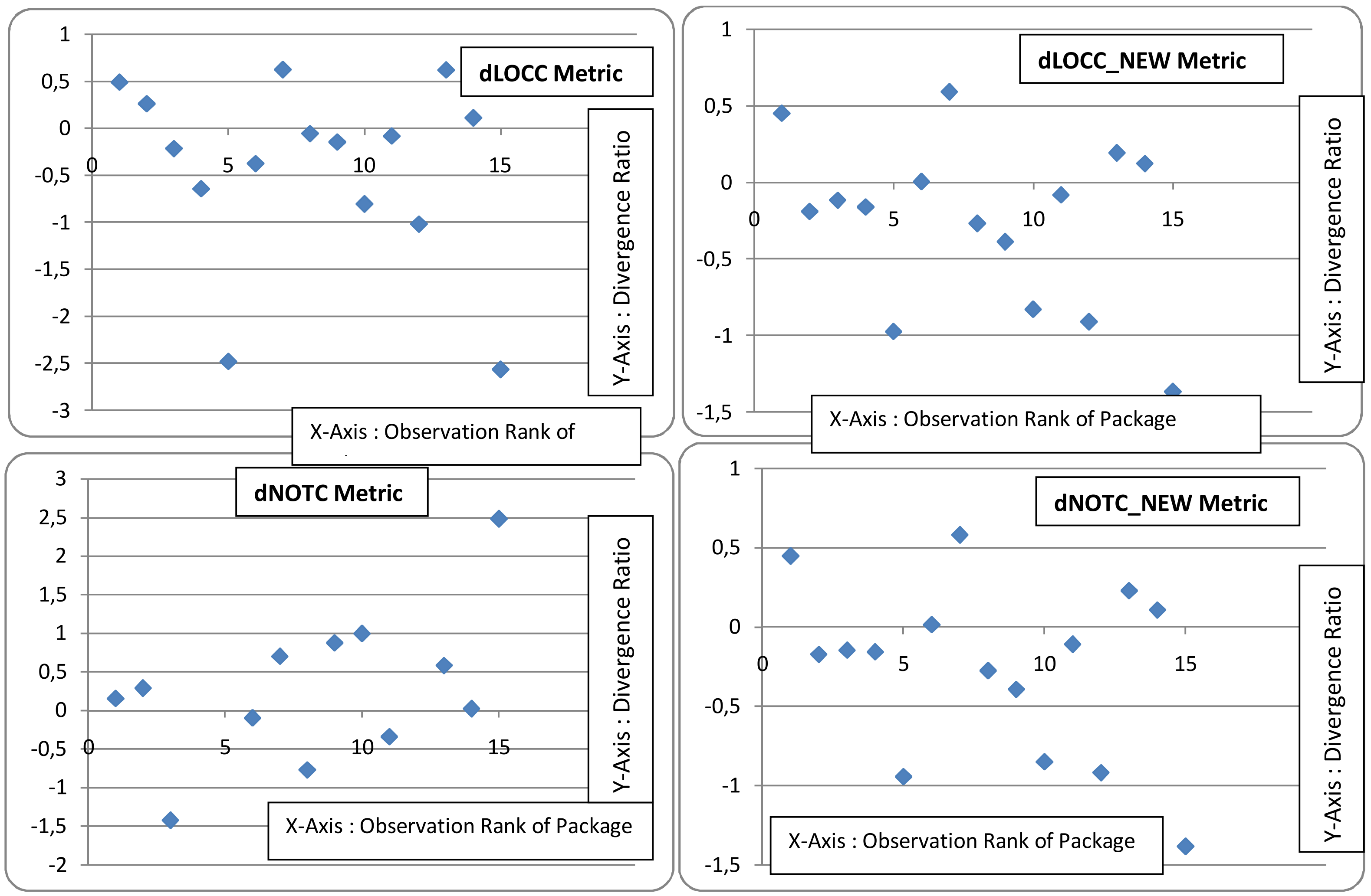


Figure 36 : Scatter Chart of Deviation – Test Metrics - Class Level – Apache ODE

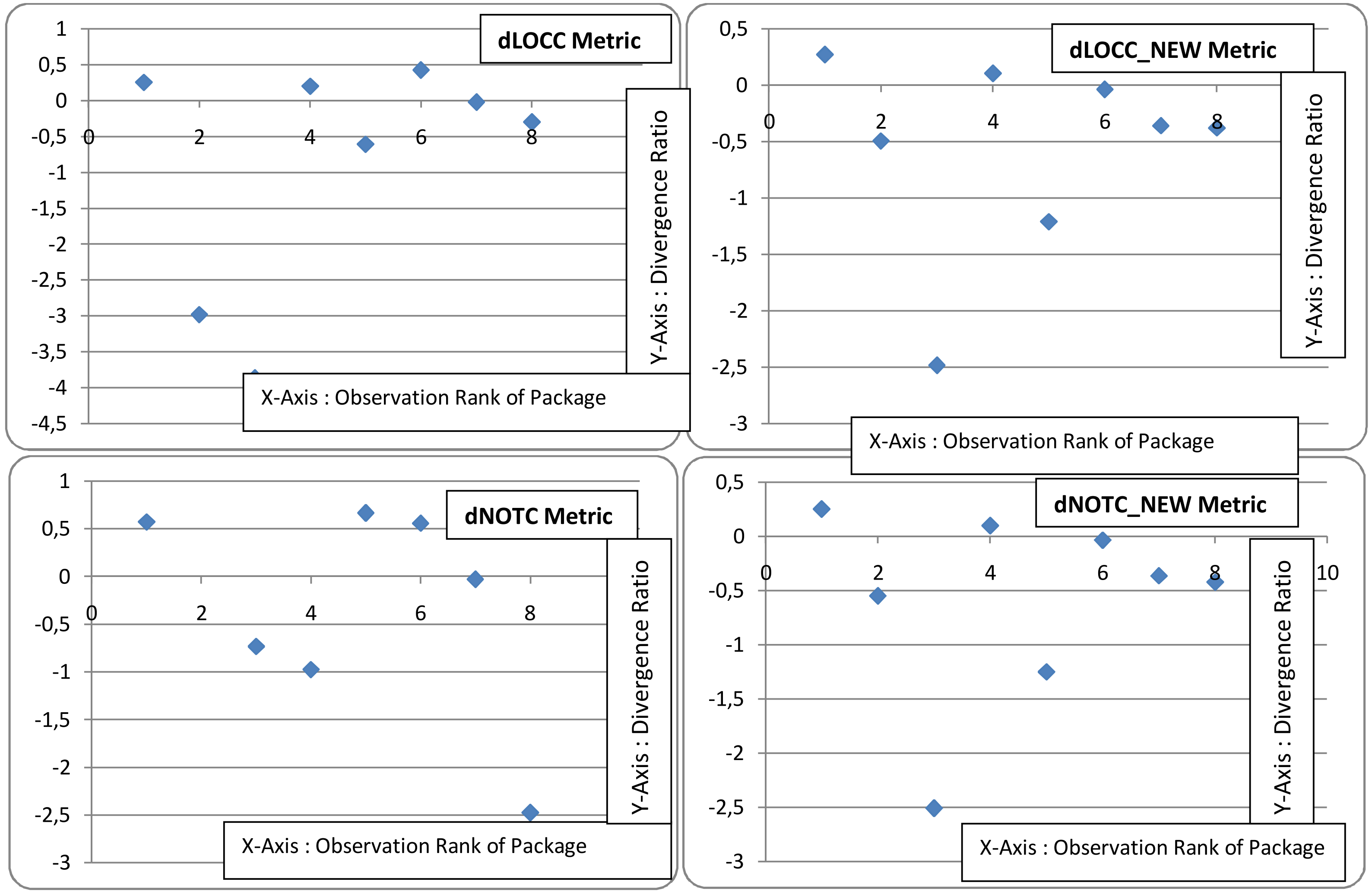


Figure 37 : Scatter Chart of Deviation – Test Metrics - Class Level – Apache OJB

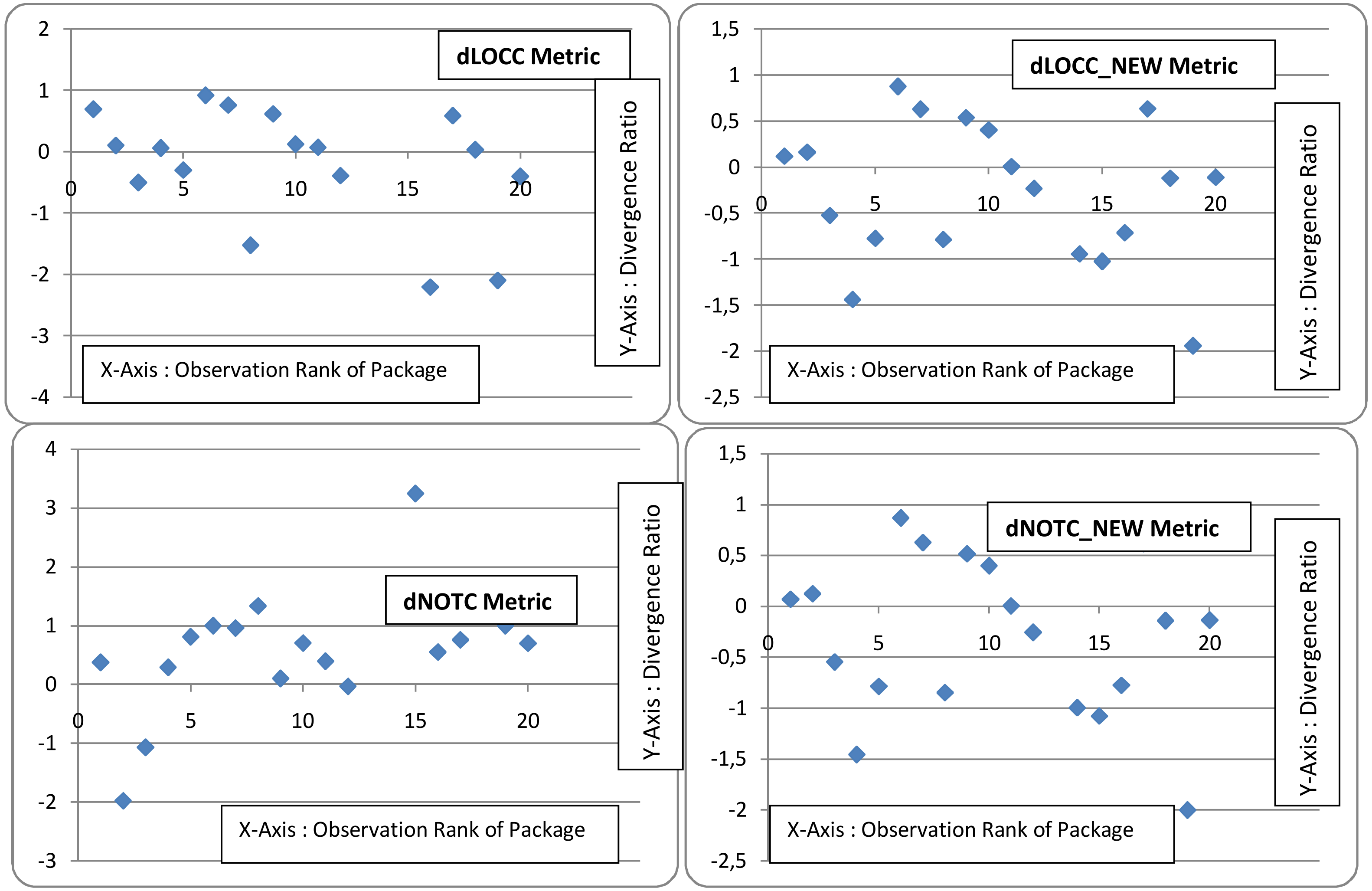


Figure 38 : Scatter Chart of Deviation – Test Metrics - Class Level – Apache OpenEJB

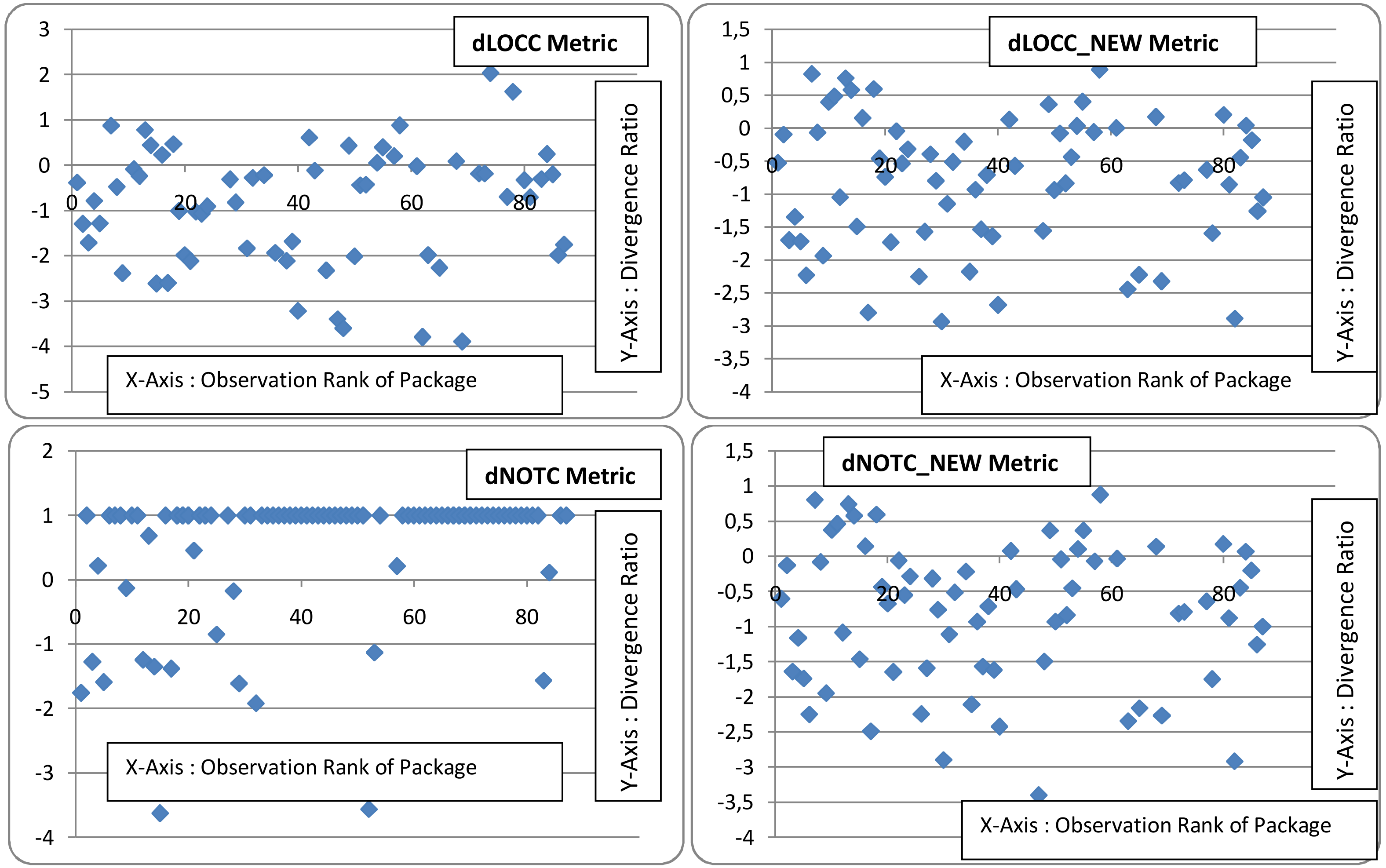


Figure 39 : Scatter Chart of Deviation – Test Metrics - Class Level – Apache Struts

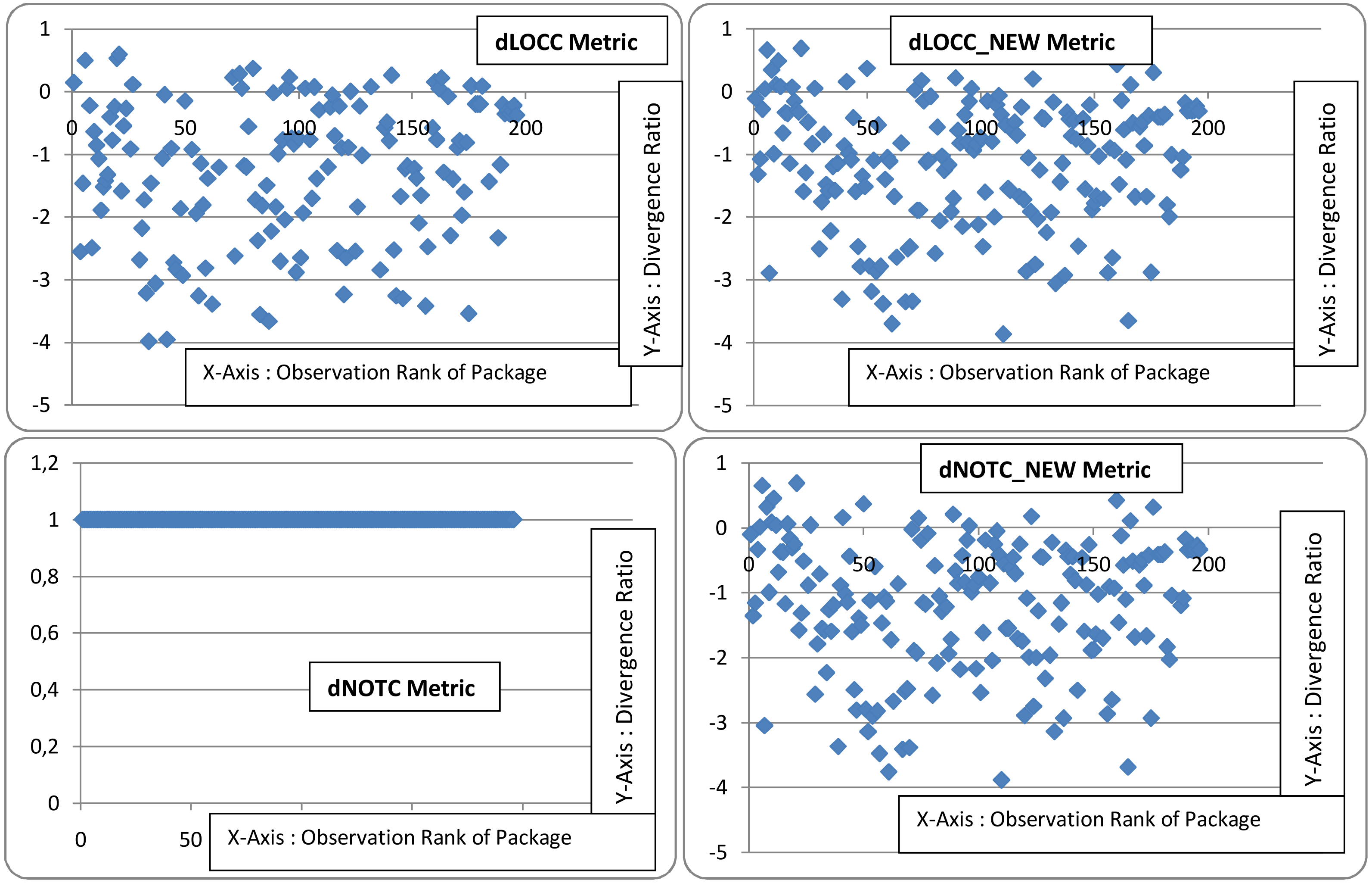


Figure 40 : Scatter Chart of Deviation – Test Metrics - Class Level – Apache Tapestry

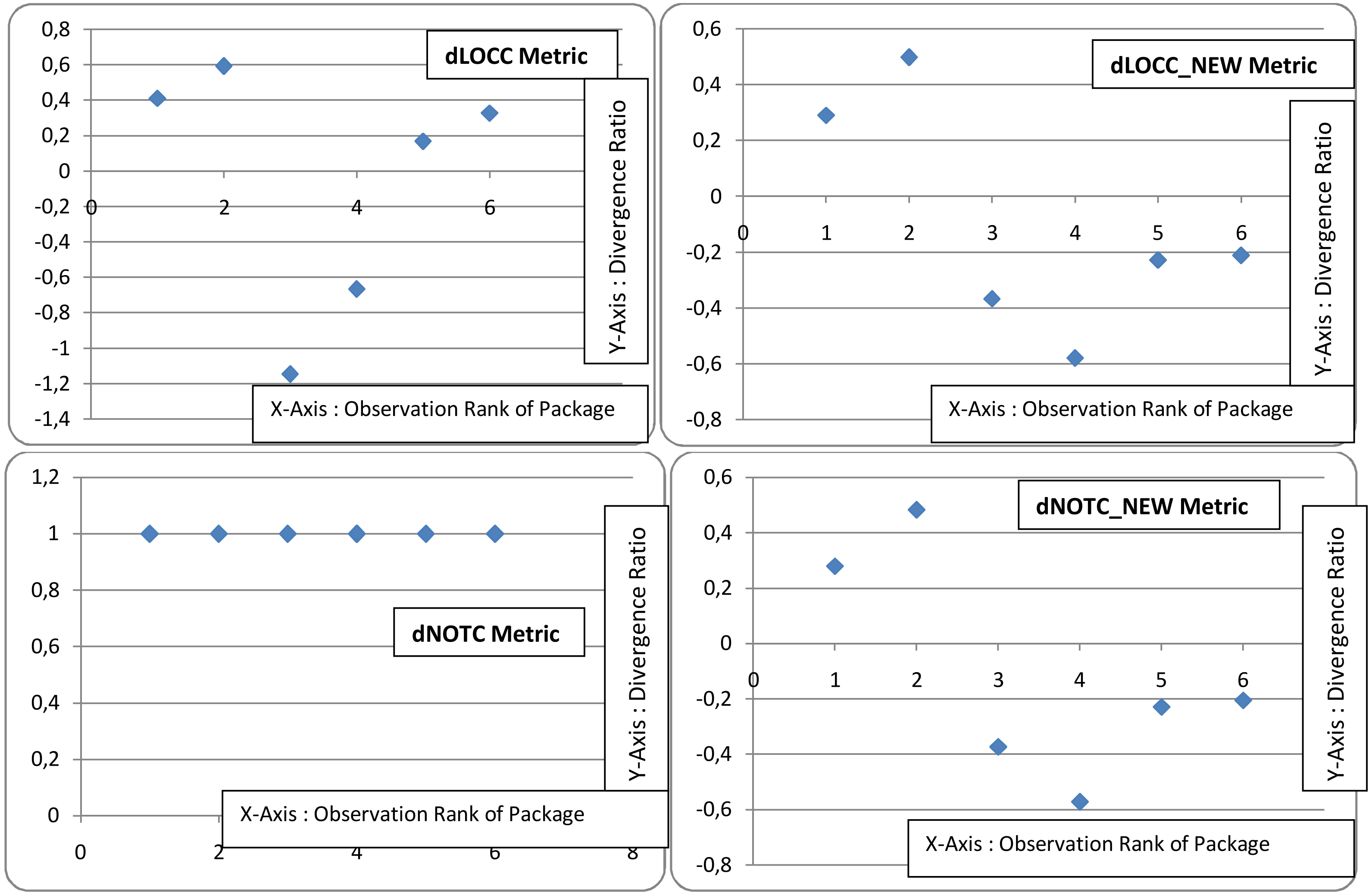


Figure 41 : Scatter Chart of Deviation – Test Metrics - Class Level – JBoss ESB

APPENDIX D. Class-Level Metrics - Regression Details

Table 53 : Correlation Results of Apache Ant – Class Level Metrics

Correlation Coefficients ρ(x; y)	dLOC_CLS	dLOC_CLS_NEW	dNOTC_CLS	dNOTC_CLS _NEW
DIT	0.28015996	0.347106414	0.071528792	0.346279932
FOUT	0.465335465	0.540516414	0.268604229	0.537871905
LCOM	0.381708981	0.442594105	0.097599529	0.441539482
LOC_CLS	0.529216752	0.590685442	0.340044596	0.589093784
LOC_CLS_NEW	0.548039915	0.608880261	0.344798362	0.60780736
NOF	0.432281884	0.48613698	0.179587047	0.48552055
NOM	0.509555664	0.558150875	0.274115192	0.558742032
NORM	0.260715393	0.290797651	0.120036016	0.290729557
NSC	0.222128229	0.205169023	0.125067824	0.206387788
NSF	0.24290609	0.277731129	0.172339751	0.275539768
NSM	0.239702277	0.247663417	0.282709837	0.244679948
RFC	0.528151399	0.595307828	0.315844765	0.593886096
SIX	0.130827422	0.150811337	0.024230257	0.150369786
TNOF	0.470858647	0.518856543	0.225704252	0.51811843
TNOM	0.527118745	0.565421019	0.336116017	0.566033736
WMC	0.550986992	0.607836091	0.35765309	0.606915291

Table 54 : Significance Results of Apache Ant – Class Level Metrics

Significance Values p(x; y)	dLOC_CLS	dLOC_CLS_NEW	dNOTC_CLS	dNOTC_CLS _NEW
DIT	6.39902E-05	5.44471E-07	0.316628647	5.81646E-07
FOUT	4.93961E-12	2.04667E-16	0.000130126	3.04955E-16
LCOM	2.8789E-08	6.63146E-11	0.171337195	7.44595E-11
LOC_CLS	1.09795E-15	5.20915E-20	9.51536E-07	6.92594E-20
LOC_CLS_NEW	6.45559E-17	1.78961E-21	6.54443E-07	2.19636E-21
NOF	2.02404E-10	3.85909E-13	0.011353365	4.17237E-13
NOM	1.76615E-14	1.30728E-17	9.31259E-05	1.18874E-17
NORM	0.000207482	3.23653E-05	0.092093174	3.25097E-05
NSC	0.001659628	0.003736203	0.079153729	0.003531669
NSF	0.000564462	7.44809E-05	0.015187575	8.53112E-05
NSM	0.000670718	0.000435086	5.44799E-05	0.000512549
RFC	1.28224E-15	2.25735E-20	5.82175E-06	2.92362E-20
SIX	0.066187265	0.03393758	0.734729551	0.0344699
TNOF	2.55245E-12	4.85286E-15	0.001387737	5.38453E-15
TNOM	1.48958E-15	4.00797E-18	1.29045E-06	3.62305E-18
WMC	4.07564E-17	2.18437E-21	2.305E-07	2.60256E-21

Table 55 : Correlation Results of Apache Lucene – Class Level Metrics

Correlation Coefficients ρ(x; y)	dLOC_CLS	dLOC_CLS_NEW	dNOTC_CLS	dNOTC_CLS _NEW
DIT	-0.027303125	-0.036054038	-0.05281954	-0.028508086
FOUT	0.408390513	0.409138686	0.31648802	0.405651577
LCOM	0.405379984	0.421607714	0.406127168	0.418429724
LOC_CLS	0.560927123	0.568145092	0.530347349	0.559618901
LOC_CLS_NEW	0.572845135	0.588152488	0.51690324	0.583047084
NOF	0.396785528	0.416722771	0.433162538	0.415334139
NOM	0.515794903	0.544812233	0.40039563	0.545510541
NORM	0.117248717	0.110179358	0.107266265	0.1138478
NSC	0.208066577	0.221375975	0.033468604	0.225631126
NSF	0.209630849	0.230530511	0.075443935	0.21757956
NSM	0.059706682	0.054054782	0.137580502	0.057616218
RFC	0.492865453	0.51386793	0.422073455	0.513838182
SIX	0.102867577	0.09010046	0.10499071	0.093487429
TNOF	0.418196941	0.442121465	0.372152354	0.435664133
TNOM	0.509642034	0.538318535	0.417334663	0.539483398
WMC	0.560135305	0.566624945	0.543727797	0.562165544

Table 56 : Significance Results of Apache Lucene – Class Level Metrics

Significance Values p(x; y)	dLOC_CLS	dLOC_CLS_NEW	dNOTC_CLS	dNOTC_CLS _NEW
DIT	0.765303276	0.693390194	0.563393404	0.75526556
FOUT	3.01709E-06	2.88219E-06	0.000383069	3.56371E-06
LCOM	3.62274E-06	1.32286E-06	3.46251E-06	1.6181E-06
LOC_CLS	1.82076E-11	8.77516E-12	3.32829E-10	2.0744E-11
LOC_CLS_NEW	5.40352E-12	1.05483E-12	1.09214E-09	1.83625E-12
NOF	6.04841E-06	1.80149E-06	6.24852E-07	1.96505E-06
NOM	1.20179E-09	8.73193E-11	4.88529E-06	8.17245E-11
NORM	0.198385974	0.226999304	0.239597211	0.211808181
NSC	0.021462899	0.014266176	0.714387637	0.012460307
NSF	0.020480655	0.010631426	0.408858016	0.016065658
NSM	0.513578373	0.554292117	0.130743026	0.528456102
RFC	8.07005E-09	1.41811E-09	1.28415E-06	1.42173E-09
SIX	0.259528277	0.323664107	0.249770853	0.305732623
TNOF	1.64202E-06	3.42605E-07	2.43354E-05	5.29242E-07
TNOM	2.03138E-09	1.60471E-10	1.73359E-06	1.44013E-10
WMC	1.97044E-11	1.02484E-11	9.67473E-11	1.60844E-11

Table 57 : Correlation Results of Apache Geronimo – Class Level Metrics

Correlation Coefficients ρ(x; y)	**dLOC_CLS**	**dLOC_CLS_NEW**	**dNOTC_CLS**	**dNOTC_CLS _NEW**
DIT	-0.286638393	-0.278976818	-0.192471255	-0.282822538
FOUT	0.503288744	0.518960674	0.169798618	0.518879926
LCOM	0.361208163	0.353491297	0.115108653	0.353099235
LOC_CLS	0.578818478	0.572190784	0.230053558	0.574035243
LOC_CLS_NEW	0.549027127	0.539021706	0.213345522	0.541205734
NOF	0.458664553	0.436714444	0.094487081	0.437643597
NOM	0.34379896	0.33432976	0.076914088	0.336178068
NORM	-0.178030487	-0.180532211	-0.073096366	-0.179872853
NSC	-0.02196932	-0.03570964	-0.062886559	-0.035844733
NSF	0.264840438	0.231209689	0.136252523	0.233179527
NSM	0.322490957	0.290029334	0.187829812	0.291514613
RFC	0.463112688	0.462766347	0.195761726	0.464865861
SIX	-0.224731444	-0.221701952	-0.115086918	-0.221191724
TNOF	0.482325357	0.447146975	0.168484654	0.449081303
TNOM	0.463429319	0.436782433	0.205387553	0.440116059
WMC	0.512015933	0.494654026	0.204615819	0.496919986

Table 58 : Significance Results of Apache Geronimo – Class Level Metrics

Significance Values p(x; y)	**dLOC_CLS**	**dLOC_CLS_NEW**	**dNOTC_CLS**	**dNOTC_CLS _NEW**
DIT	0.004022259	0.005169536	0.056309902	0.004561618
FOUT	1.0993E-07	3.7314E-08	0.092910055	3.75276E-08
LCOM	0.000239573	0.000331784	0.256565276	0.000337244
LOC_CLS	3.48698E-10	6.12964E-10	0.021977864	5.2457E-10
LOC_CLS_NEW	4.00274E-09	8.62125E-09	0.033982645	7.30738E-09
NOF	1.80194E-06	6.20962E-06	0.352227115	5.90294E-06
NOM	0.000493701	0.000719164	0.449236739	0.000668874
NORM	0.077901012	0.073746442	0.47212523	0.074823718
NSC	0.829111026	0.725658936	0.536325414	0.724662572
NSF	0.008069603	0.021302264	0.178703244	0.020192159
NSM	0.001132208	0.003591572	0.062637659	0.003416294
RFC	1.38749E-06	1.41621E-06	0.052149202	1.25041E-06
SIX	0.025329547	0.027425091	0.256655694	0.027792125
TNOF	4.29704E-07	3.48604E-06	0.095504316	3.12555E-06
TNOM	1.36173E-06	6.18668E-06	0.041407782	5.15489E-06
WMC	6.06348E-08	1.94884E-07	0.042194802	1.67951E-07

Table 59 : Correlation Results of Apache Mina – Class Level Metrics

Correlation Coefficients ρ(x; y)	**dLOC_CLS**	**dLOC_CLS_NEW**	**dNOTC_CLS**	**dNOTC_CLS _NEW**
DIT	0.023379215	0.070273641	0.041970234	0.054806715
FOUT	0.47137675	0.563657758	0.411163542	0.561488651
LCOM	0.152446597	0.180997392	0.152201468	0.198265694
LOC_CLS	0.672571809	0.663715471	0.532584337	0.664936659
LOC_CLS_NEW	0.666972127	0.668498168	0.569985807	0.674603175
NOF	0.195553628	0.215088974	0.154387264	0.227334916
NOM	0.509756275	0.546355791	0.476892894	0.560168157
NORM	0.378170658	0.368166139	0.288302328	0.353249647
NSC	0.321352399	0.396466701	0.383249091	0.396466701
NSF	0.325169928	0.374600482	0.424527499	0.381830822
NSM	0.11301952	0.067668806	0.223041567	0.067668806
RFC	0.582824364	0.643613501	0.634518592	0.660445999
SIX	0.128840767	0.134818405	0.130350295	0.115702363
TNOF	0.352021995	0.371958272	0.309853075	0.384635154
TNOM	0.51635854	0.530701777	0.482032497	0.545128025
WMC	0.622286879	0.606875825	0.528421276	0.613292941

Table 60 : Significance Results of Apache Mina – Class Level Metrics

Significance Values p(x; y)	**dLOC_CLS**	**dLOC_CLS_NEW**	**dNOTC_CLS**	**dNOTC_CLS _NEW**
DIT	0.907851376	0.727611094	0.835341746	0.785996977
FOUT	0.01306501	0.002200384	0.033119708	0.002308143
LCOM	0.447788912	0.366270444	0.448528033	0.321511225
LOC_CLS	0.000121444	0.000160471	0.004237675	0.000154506
LOC_CLS_NEW	0.000144996	0.000138208	0.00191035	0.000113776
NOF	0.328308964	0.281291149	0.441960351	0.254127729
NOM	0.006604655	0.003193843	0.011899488	0.002375937
NORM	0.051778035	0.05882398	0.144743963	0.0706899
NSC	0.10216036	0.040619846	0.048464506	0.040619846
NSF	0.097922693	0.054212032	0.027305569	0.049372587
NSM	0.574605667	0.737350463	0.263448551	0.737350463
RFC	0.001421509	0.000292436	0.000378423	0.000177454
SIX	0.521863437	0.502564753	0.51695644	0.565497904
TNOF	0.071742923	0.056070358	0.115752933	0.047589809
TNOM	0.005827061	0.00440071	0.010892636	0.003276971
WMC	0.000528549	0.000789872	0.004605318	0.000669883

Table 61 : Correlation Results of Apache Wicket – Class Level Metrics

Correlation Coefficients ρ(x; y)	dLOC_CLS	dLOC_CLS_NEW	dNOTC_CLS	dNOTC_CLS _NEW
DIT	-0.098380836	-0.119824277	-0.191061753	-0.112085878
FOUT	0.281636843	0.291685119	0.313167175	0.299733656
LCOM	0.304876502	0.310947015	0.280949355	0.313646848
LOC_CLS	0.404702465	0.382384602	0.411272177	0.389782246
LOC_CLS_NEW	0.419445226	0.398654604	0.420789779	0.407561675
NOF	0.30823495	0.310944474	0.246952352	0.312079648
NOM	0.372527459	0.3470549	0.302075826	0.354344525
NORM	0.057424865	0.016543604	-0.005128961	0.016884093
NSC	0.020767397	0.014359867	-0.008301737	0.012493019
NSF	0.138610343	0.124466491	0.20719539	0.133992599
NSM	0.143583437	0.172076599	0.253788202	0.180184462
RFC	0.356519995	0.331162657	0.37931637	0.3412425
SIX	-0.021298908	-0.059410543	-0.105544773	-0.060616129
TNOF	0.304467304	0.299036118	0.345037764	0.306901551
TNOM	0.42670423	0.414496335	0.411548494	0.422802268
WMC	0.434872088	0.419474402	0.442162449	0.426835867

Table 62 : Significance Results of Apache Wicket – Class Level Metrics

Significance Values p(x; y)	dLOC_CLS	dLOC_CLS_NEW	dNOTC_CLS	dNOTC_CLS _NEW
DIT	0.322823657	0.22797509	0.053205466	0.259649152
FOUT	0.003951631	0.002794839	0.001277727	0.002098786
LCOM	0.00174035	0.001389158	0.004044573	0.001254747
LOC_CLS	2.23493E-05	6.71959E-05	1.5918E-05	4.70611E-05
LOC_CLS_NEW	1.03319E-05	3.03543E-05	9.61234E-06	1.92976E-05
NOF	0.001537221	0.001389291	0.011910541	0.001331253
NOM	0.000106606	0.000328727	0.001928005	0.000240457
NORM	0.564505763	0.868265946	0.958992176	0.865579842
NSC	0.835060111	0.885527687	0.933671357	0.90032738
NSF	0.162617143	0.210330169	0.035730067	0.177216971
NSM	0.147914417	0.082199697	0.009689087	0.068562422
RFC	0.000218712	0.000633472	7.76999E-05	0.000419554
SIX	0.830900758	0.55109538	0.288664144	0.543029409
TNOF	0.001766694	0.002152225	0.000357963	0.001615144
TNOM	6.97138E-06	1.34407E-05	1.56899E-05	8.62289E-06
WMC	4.42887E-06	1.03158E-05	2.92461E-06	6.92124E-06

Table 63 : Correlation Results of JBoss Cache – Class Level Metrics

Correlation Coefficients ρ(x; y)	**dLOC_CLS**	**dLOC_CLS_NEW**	**dNOTC_CLS**	**dNOTC_CLS _NEW**
DIT	-0.02114279	-0.037094736	-0.103770699	-0.039335358
FOUT	0.113986249	0.142226085	0.143954287	0.149626402
LCOM	-0.077495126	-0.065803677	0.036489299	-0.060567904
LOC_CLS	0.134875645	0.154781156	0.206746082	0.16599805
LOC_CLS_NEW	0.096046382	0.117778532	0.192547935	0.129392749
NOF	-0.075252984	-0.064096507	0.08800657	-0.055704594
NOM	-0.041068546	-0.021196487	0.096175771	-0.008310781
NORM	-0.238797246	-0.243471629	-0.158504331	-0.235954495
NSC	0.250884285	0.252883945	0.130441238	0.255111106
NSF	-0.147039712	-0.151491948	-0.132438935	-0.148492739
NSM	-0.100410014	-0.07201602	0.03127047	-0.069296251
RFC	-0.016852869	0.012219108	0.065854449	0.024038898
SIX	-0.202313468	-0.224309282	-0.167219641	-0.218591109
TNOF	-0.146554218	-0.140046827	0.014882407	-0.129955897
TNOM	-0.059763864	-0.037840677	0.086031185	-0.025773217
WMC	0.109695722	0.134400786	0.215838381	0.145720638

Table 64 : Significance Results of JBoss Cache – Class Level Metrics

Significance Values p(x; y)	**dLOC_CLS**	**dLOC_CLS_NEW**	**dNOTC_CLS**	**dNOTC_CLS _NEW**
DIT	0.865149103	0.765686363	0.403338121	0.751974495
FOUT	0.358379407	0.250923164	0.245151334	0.226849785
LCOM	0.533070268	0.596761117	0.769403976	0.626338275
LOC_CLS	0.276502384	0.211062678	0.093223786	0.179428217
LOC_CLS_NEW	0.439422676	0.342505969	0.118509687	0.296675811
NOF	0.545019922	0.606336764	0.478830896	0.654348831
NOM	0.741419909	0.864809829	0.43880371	0.94678263
NORM	0.051641413	0.047101893	0.200153565	0.054572811
NSC	0.040578826	0.038953324	0.29274558	0.037206855
NSF	0.235074301	0.221044013	0.28535219	0.230429302
NSM	0.418819848	0.562497426	0.801656129	0.577383689
RFC	0.892327061	0.921821834	0.596477363	0.846887095
SIX	0.100616957	0.068027053	0.176203145	0.075546333
TNOF	0.236640642	0.258332538	0.904854008	0.294560602
TNOM	0.630934405	0.76111325	0.488793912	0.835988677
WMC	0.376873568	0.278212508	0.079395066	0.239346787

Table 65 : Correlation Results of JBoss Drools – Class Level Metrics

Correlation Coefficients ρ(x; y)	**dLOC_CLS**	**dLOC_CLS_NEW**	**dNOTC_CLS**	**dNOTC_CLS _NEW**
DIT	0.46939014	0.45525277	0.386842181	0.455193291
FOUT	0.559375165	0.580511262	0.439156437	0.580598009
LCOM	0.170985554	0.15995913	0.187220989	0.157585682
LOC_CLS	0.575394228	0.577278558	0.479771517	0.575087767
LOC_CLS_NEW	0.592558914	0.595892966	0.497946638	0.593491767
NOF	0.294304907	0.281596125	0.236853876	0.27927107
NOM	0.504539237	0.503758471	0.398136387	0.501472489
NORM	0.279399096	0.257638747	0.16998289	0.256403447
NSC	0.067624489	0.061415013	0.160034063	0.057169221
NSF	0.200060673	0.183016734	0.229165298	0.181158035
NSM	-0.06900993	-0.053417283	0.104387396	-0.048326522
RFC	0.610652642	0.624282062	0.466069132	0.622917759
SIX	0.292399593	0.273358823	0.188138911	0.272660934
TNOF	0.35383294	0.335739462	0.342108771	0.333048699
TNOM	0.505789362	0.511864075	0.442604485	0.510218663
WMC	0.553133008	0.552740987	0.501626918	0.550303329

Table 66 : Significance Results of JBoss Drools – Class Level Metrics

Significance Values p(x; y)	**dLOC_CLS**	**dLOC_CLS_NEW**	**dNOTC_CLS**	**dNOTC_CLS _NEW**
DIT	2.59234E-09	8.76012E-09	1.53711E-06	8.80409E-09
FOUT	2.60833E-13	1.96121E-14	3.28134E-08	1.93974E-14
LCOM	0.039754871	0.054620115	0.024139359	0.058357245
LOC_CLS	3.73262E-14	2.94888E-14	1.02276E-09	3.87792E-14
LOC_CLS_NEW	4.11977E-15	2.64439E-15	1.85899E-10	3.64102E-15
NOF	0.000326715	0.000600866	0.004126265	0.00066967
NOM	9.76697E-11	1.05481E-10	7.07481E-07	1.31986E-10
NORM	0.0006657	0.001756895	0.040948646	0.001851989
NSC	0.418984379	0.463054149	0.054505422	0.494600997
NSF	0.015837355	0.027567683	0.00556054	0.029211011
NSM	0.409495817	0.523396945	0.211471976	0.563784652
RFC	3.48791E-16	4.8807E-17	3.46804E-09	5.96841E-17
SIX	0.000358614	0.000878546	0.023441547	0.000906798
TNOF	1.26586E-05	3.65409E-05	2.5348E-05	4.25444E-05
TNOM	8.63138E-11	4.70022E-11	2.48702E-08	5.54792E-11
WMC	5.41125E-13	5.66217E-13	1.30009E-10	7.49575E-13

Table 67 : Correlation Results of JBoss Richfaces – Class Level Metrics

Correlation Coefficients ρ(x; y)	dLOC_CLS	dLOC_CLS_NEW	dNOTC_CLS	dNOTC_CLS _NEW
DIT	0.007577813	-0.075541874	-0.156678985	-0.039872316
FOUT	0.326217618	0.266379155	-0.149975008	0.294183956
LCOM	0.076500152	0.173313319	0.09150661	0.169036204
LOC_CLS	0.432827222	0.421621102	-0.016880105	0.441329672
LOC_CLS_NEW	0.434587737	0.444623056	-0.002231878	0.462689646
NOF	0.053974328	0.154860302	0.095486667	0.142567642
NOM	0.318332318	0.390525482	-0.072655641	0.389355176
NORM	0.101943862	0.017360312	-0.172969551	0.044882136
NSC	0.091969894	0.081058114	-0.232666727	0.112781353
NSF	0.059673046	0.0366974	-0.260781742	0.043266688
NSM	0.03566422	-0.011404722	0.214039675	-0.003085047
RFC	0.337746174	0.332272465	0.061403743	0.357952953
SIX	0.085738315	-0.002085839	-0.19324947	0.030982336
TNOF	0.118010149	0.179841413	-0.040062728	0.173630273
TNOM	0.333765645	0.40299728	-0.030604865	0.403868054
WMC	0.440307243	0.460169426	-0.013104161	0.473883085

Table 68 : Significance Results of JBoss Richfaces – Class Level Metrics

Significance Values p(x; y)	dLOC_CLS	dLOC_CLS_NEW	dNOTC_CLS	dNOTC_CLS _NEW
DIT	0.948552643	0.519481354	0.179467293	0.734127816
FOUT	0.004286663	0.020884106	0.199042477	0.01041079
LCOM	0.514181717	0.137013188	0.43491992	0.147124263
LOC_CLS	0.000105163	0.000165051	0.885705089	7.39118E-05
LOC_CLS_NEW	9.78327E-05	6.43123E-05	0.984837907	2.92145E-05
NOF	0.645576561	0.18463347	0.415121551	0.222397148
NOM	0.005380315	0.000532444	0.535608382	0.000555222
NORM	0.384135557	0.882476429	0.137805921	0.702198026
NSC	0.432588284	0.489356767	0.044561659	0.335352597
NSF	0.611062469	0.754602651	0.023836215	0.712441
NSM	0.761302938	0.922637432	0.065192343	0.979043097
RFC	0.003042125	0.003585845	0.600741388	0.001615605
SIX	0.464539327	0.985829911	0.09667744	0.791878836
TNOF	0.313281055	0.122605097	0.732905564	0.136285347
TNOM	0.003429542	0.000337432	0.794358376	0.000326636
WMC	7.71516E-05	3.27015E-05	0.911153233	1.7514E-05

APPENDIX E. Package-Level Metrics - Regression Details

Table 69 : Correlation & Significance Results of Apache Ant – Package Level Metrics

Correlation Coefficients ρ(x; y)	**dLOC_PKG**	**dNOTC_PKG**
CA	0.519851187	0.326803381
CE	0.567261655	0.521223784
NOC	0.664882124	0.574703021
NOI	0.559597873	0.396221904
RMA	0.291128883	0.086692298
RMD	0.389641753	0.251756499
RMI	-0.396489208	-0.161823604
LOC_PKG	0.660203911	0.561851518

Significance Values p(x; y)	**dLOC_PKG**	**dNOTC_PKG**
CA	0.001930633	0.003497073
CE	0.000576376	9.94511E-07
NOC	2.43606E-05	3.74857E-08
NOI	0.000709423	0.000329412
RMA	0.100228101	0.450424976
RMD	0.024997904	0.026182052
RMI	0.022349973	0.156933278
LOC_PKG	2.90919E-05	8.69075E-08

Table 70 : Correlation & Significance Results of Apache Lucene – Package Level Metrics

Correlation Coefficients ρ(x; y)	**dLOC_PKG**	**dNOTC_PKG**
CA	0.287382154	0.292004426
CE	0.596051245	0.575956561
NOC	0.733938578	0.574325511
NOI	0.155220854	0.322887716
RMA	-0.025437044	0.256455165
RMD	0.18627775	0.161211775
RMI	-0.121178245	-0.092396808
LOC_PKG	0.598901189	0.577054656

Significance Values p(x; y)	**dLOC_PKG**	**dNOTC_PKG**
CA	0.030191382	0.007770067
CE	9.98532E-07	1.50223E-08
NOC	8.28418E-11	1.68496E-08
NOI	0.24893789	0.003089712
RMA	0.851016886	0.020035147
RMD	0.165324095	0.147923174
RMI	0.369227086	0.40902419
LOC_PKG	8.5944E-07	1.39001E-08

Table 71 : Correlation & Significance Results of Apache Geronimo – Package Level Metrics

Correlation Coefficients $\rho(x; y)$	**dLOC_PKG**	**dNOTC_PKG**
CA	0.236553046	0.211064841
CE	0.568573025	0.448951485
NOC	0.592423244	0.417613043
NOI	0.266118576	0.15724004
RMA	0.189777708	0.068150629
RMD	0.083536889	0.029898006
RMI	-0.166727532	-0.055135373
LOC_PKG	0.521186982	0.412977904

Significance Values $p(x; y)$	**dLOC_PKG**	**dNOTC_PKG**
CA	0.019022144	0.000404913
CE	1.01351E-09	3.82103E-15
NOC	1.31236E-10	4.07153E-13
NOI	0.008083095	0.008754665
RMA	0.061255622	0.258291647
RMD	0.413475267	0.620273613
RMI	0.100828967	0.360621491
LOC_PKG	3.75175E-08	7.8025E-13

Table 72 : Correlation & Significance Results of Apache Mina – Package Level Metrics

Correlation Coefficients $\rho(x; y)$	**dLOC_PKG**	**dNOTC_PKG**
CA	0.505874008	0.503569578
CE	0.493268486	0.5765248
NOC	0.551700892	0.648523972
NOI	0.014366571	0.01778955
RMA	0.321757314	-0.003555213
RMD	0.125129533	0.267226813
RMI	-0.372306443	-0.400581111
LOC_PKG	0.630546956	0.653804779

Significance Values $p(x; y)$	**dLOC_PKG**	**dNOTC_PKG**
CA	0.03220275	0.000924911
CE	0.037512884	9.87981E-05
NOC	0.017615379	6.02832E-06
NOI	0.954880676	0.913240737
RMA	0.192898635	0.982629685
RMD	0.620798376	0.095520817
RMI	0.128140689	0.010424461
LOC_PKG	0.005025142	4.76988E-06

Table 73 : Correlation & Significance Results of Apache Wicket – Package Level Metrics

Correlation Coefficients $\rho(x; y)$	**dLOC_PKG**	**dNOTC_PKG**
CA	0.63607228	0.361187162
CE	0.541879375	0.395815996
NOC	0.606088724	0.429876537
NOI	0.556027056	0.228047848
RMA	0.408208792	0.056611443
RMD	0.442117495	0.224982321
RMI	-0.578463409	-0.234482127
LOC_PKG	0.639945691	0.409634449

Significance Values $p(x; y)$	**dLOC_PKG**	**dNOTC_PKG**
CA	1.24491E-11	8.31424E-09
CE	2.90501E-08	1.99591E-10
NOC	1.93906E-10	3.25859E-12
NOI	1.05431E-08	0.000368741
RMA	5.90252E-05	0.382585477
RMD	1.14639E-05	0.000444161
RMI	1.91128E-09	0.000247492
LOC_PKG	8.54154E-12	3.97482E-11

Table 74 : Correlation & Significance Results of JBoss Cache – Package Level Metrics

Correlation Coefficients $\rho(x; y)$	dLOC_PKG	dNOTC_PKG
CA	0.423910234	0.156765954
CE	0.648395451	0.646551671
NOC	0.71363301	0.661003199
NOI	0.494348754	0.337248205
RMA	-0.17221122	0.066887336
RMD	0.21483022	-0.035131822
RMI	-0.25019023	0.132578562
LOC_PKG	0.610964285	0.530935025

Significance Values $p(x; y)$	dLOC_PKG	dNOTC_PKG
CA	0.004614398	0.231629281
CE	2.58191E-06	2.40061E-08
NOC	7.80548E-08	9.0298E-09
NOI	0.000753285	0.008412596
RMA	0.26947891	0.611609439
RMD	0.166530325	0.789863675
RMI	0.105637931	0.312585116
LOC_PKG	1.35306E-05	1.27634E-05

Table 75 : Correlation & Significance Results of JBoss Drools – Package Level Metrics

Correlation Coefficients $\rho(x; y)$	dLOC_PKG	dNOTC_PKG
CA	0.307687981	0.308000819
CE	0.405116405	0.574364244
NOC	0.465860501	0.577507081
NOI	0.228808587	0.241666487
RMA	0.088252726	-0.066997804
RMD	0.178655205	0.057749734
RMI	-0.198164183	0.048911058
LOC_PKG	0.432348367	0.552754978

Significance Values $p(x; y)$	dLOC_PKG	dNOTC_PKG
CA	0.021062185	0.00139169
CE	0.001953558	1.4915E-10
NOC	0.000296536	1.12259E-10
NOI	0.089845816	0.013005024
RMA	0.517766982	0.497094295
RMD	0.187702265	0.558437628
RMI	0.143182653	0.620256786
LOC_PKG	0.000875876	9.72481E-10

Table 76 : Correlation & Significance Results of JBoss Richfaces – Package Level Metrics

Correlation Coefficients $\rho(x; y)$	dLOC_PKG	dNOTC_PKG
CA	0.262958707	0.145839686
CE	0.437716233	0.075605144
NOC	0.57195543	0.163815157
NOI	0.283530056	0.069372151
RMA	0.172146809	-0.0330483
RMD	-0.101935024	0.044394785
RMI	-0.001005955	-0.034136762
LOC_PKG	0.424208096	0.126432455

Significance Values $p(x; y)$	dLOC_PKG	dNOTC_PKG
CA	0.096700584	0.14766897
CE	0.004209306	0.454691234
NOC	9.35899E-05	0.103402427
NOI	0.072431002	0.492822452
RMA	0.28181599	0.744108735
RMD	0.525968037	0.660963958
RMI	0.995019594	0.735985675
LOC_PKG	0.005707482	0.210036258

APPENDIX F. Descriptive Statistics of Metrics Used

Table 77 : Descriptive Statistics of Class Metrics – All Projects As One Single Project

All As One	Average	Standard Deviation	Maximum
DIT	1.89	1.29	6.59
FOUT	4.28	5.57	53.25
LCOM	0.22	0.33	1.35
LOCC	59.28	133.29	2227
NOF	2.11	3.76	46.4
NOM	7.07	10.72	150.41
NORM	0.50	1.47	22.2
NSC	0.42	2.81	73.7
NSF	1.02	4.15	89.2
NSM	0.55	2.24	38.75
RFC	16.66	22.53	260.9
SIX	0.20	0.53	4.3
TNOF	3.13	6.11	97.32
TNOM	7.62	10.97	153.7
WMC	16.26	37.8	748
dLOC_CLS	72.44	99.41	1156
dLOC_CLS_NEW	52.99	1.73	604.50
dNOTC_CLS	9.40	0.63	307
dNOTC_CLS NEW	36.90	1.21	423.74

Table 78 : Descriptive Statistics of Class Metrics – Apache Projects Only

Apache Only	Average	Standard Deviation	Maximum
DIT	2.01	1.35	6.55
FOUT	4.27	5.59	52.85
LCOM	0.23	0.33	1.32
LOCC	60.77	130	2043
NOF	2.20	3.93	47.33
NOM	7.24	10.95	150.92
NORM	0.52	1.46	20.81
NSC	0.47	3.05	70.29
NSF	1.05	3.80	71.44
NSM	0.55	2.25	34.52
RFC	17.37	23.58	270
SIX	0.22	0.55	4.27
TNOF	3.25	5.91	80.63
TNOM	7.78	11.19	152.8
WMC	16.92	39.06	789.7
dLOC_CLS	66.34	2.87	851
dLOC_CLS_NEW	50.10	1.82	462.40
dNOTC_CLS	7.66	0.63	307
dNOTC_CLS NEW	34.87	1.26	324.38

Table 79 : Descriptive Statistics of Class Metrics – JBoss Projects Only

JBoss Only	**Average**	**Standard Deviation**	**Maximum**
DIT	1.70	1.19	6.65
FOUT	4.28	5.52	53.88
LCOM	0.21	0.32	1.39
LOCC	57	138	2519
NOF	1.96	3.49	44.94
NOM	6.81	10.34	149.58
NORM	0.47	1.47	24.4
NSC	0.35	2.44	79.06
NSF	0.98	4.71	117.35
NSM	0.54	2.21	45.47
RFC	15.53	20.86	245.53
SIX	0.17	0.49	4.35
TNOF	2.94	6.42	123
TNOM	7.35	10.61	155
WMC	15.22	35.75	681
dLOC_CLS	93.47	7.87	1156
dLOC_CLS_NEW	62.95	4.46	604.50
dNOTC_CLS	15.40	1.71	218
dNOTC_CLS NEW	43.87	3.12	423.74

Table 80 : Descriptive Statistics of Package Metrics – All Projects As One Single Project

All As One	Average	Standard Deviation	Maximum
CA	16.56	43.16	344
CE	7.75	11.88	75.98
NOC	10.81	16.89	109
NOI	1.49	3.48	23.63
RMA	0.17	0.25	1
RMD	0.30	0.28	1
RMI	0.60	0.34	1
dNOTC_PKG	19.45	67	518
LOC_PKG	715	1421	9764
dLOC_PKG	449	847	3301

Table 81 : Descriptive Statistics of Package Metrics – Apache Projects Only

Apache Only	Average	Standard Deviation	Maximum
CA	17.85	43.77	307.04
CE	8.67	13.95	90.3
NOC	11.69	18.86	120
NOI	1.70	4.08	27.14
RMA	0.18	0.24	1
RMD	0.30	0.28	1
RMI	0.59	0.34	1
dNOTC_PKG	18.16	65.51	468
LOC_PKG	757	1491	10266
dLOC_PKG	416	795	3159

Table 82 : Descriptive Statistics of Package Metrics – JBoss Projects Only

JBoss Only	Average	Standard Deviation	Maximum
CA	14.51	42.20	402.94
CE	6.28	8.59	53.29
NOC	9.41	13.75	91.7
NOI	1.16	2.52	18.06
RMA	0.17	0.25	1
RMD	0.31	0.29	1
RMI	0.61	0.34	1
dNOTC_PKG	21.75	71.17	607
LOC_PKG	644	1301	8917
dLOC_PKG	504	932	3539

Table 83 : Descriptive Statistics of Method Metrics – All Projects As One Single Project

All As One	Average	Standard Deviation	Maximum
MLOC	7.61	16.89	541
NBD	1.45	0.91	9.72
PAR	0.87	1.14	13.52
VG	2.09	3.67	125

Table 84 : Descriptive Statistics of Method Metrics – Apache Projects Only

Apache Only	Average	Standard Deviation	Maximum
MLOC	7.52	16.89	589
NBD	1.46	0.90	9.61
PAR	0.90	1.15	13
VG	2.11	3.63	109

Table 85 : Descriptive Statistics of Method Metrics – JBoss Projects Only

JBoss Only	Average	Standard Deviation	Maximum
MLOC	7.76	16.90	466
NBD	1.45	0.92	9.88
PAR	0.83	1.12	14
VG	2.07	3.74	151

APPENDIX G. Definitions of Software Design Paramaters

Coupling

IEEE [16] defines coupling as “The manner and degree of interdependence between software modules”. In an object-oriented design, coupling refers to relationships and dependencies between the communicating modules. Classes (objects) are said to be coupled when [33]:

- A message is passed between objects,.
- Methods declared in one class use methods or attributes of the other classes.
- Superclasses and their subclasses are related closely through inheritance.

A good object-oriented design is expected to have low coupling. Low coupling means, you have to have a minimal impact on the other parts of your software system, when you change one part of it. In addition, low coupling requires that you should need few modules to understand a specific module. [23]

Excessive coupling between object classes is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse it in another application. In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design, and therefore maintenance is more difficult.

A measure of coupling is useful to determine how complex the testing of various parts of a design is likely to be. The higher the inter-object class coupling, the more rigorous the testing needs to be.

High coupling makes damages on many of the quality attributes of a software system. As our focus is primarily testability and testing, we will define the effects of coupling (and the following attributes, as well) on testability.

When a class has high coupling, this mean you have to consume more resource, both time and effort, to be able to understand and test it, as you have to trace all the coupled external pieces (other coupled classes) to obtain the functionality roadmap of the class to be tested. Besides, high coupling decreases the possibility of reusability, as the components (classes or subsystems) you want to reuse will be dependent on many other components and it will be difficult to extract the required component from its context.

There are different types of coupling defined in the literature. A significant one was proposed by Timothy Budd. [8] Budd states that coupling between classes can occur due to different reasons. He identifies six types of coupling as a list, ranked from worst to better in order of acceptance and desire of occurrence with respect to others as follows:

- Internal data coupling
- Global data coupling
- Control (sequence) coupling
- Component coupling
- Parameter coupling
- Subclass coupling

Internal data coupling occurs when instances of one class is allowed to modify the local data values (instance variables) in another class. This type of coupling is strongly undesirable as it complicates the ability to understand classes in isolation.

Global data coupling occurs when two or more classes are bound together by their reliance on common global data structures. This type of coupling is also undesirable as it complicates the understanding of classes taken in isolation.

Control (sequence) coupling when one class has to perform operations in a certain fixed order, but the order is controlled elsewhere. This type of coupling is also undesirable as it indicates that the designer of a class was following a lower level of abstraction than was necessary.

Component coupling occurs when one class maintain a data field or value that is an instance of another class. The relationship in a component coupling is preferred to be one way, ideally.

Parameter coupling occurs when one class must invoke services and routines from another, and the only relationships are the number and type of parameters supplied and the type of the value returned. This type of coupling is the most benign as it is common, easy to see and to verify statistically (with tools that check parameter class against definition, for example).

Subclass coupling is particular to object-oriented environment and it describes the relationship a class has with its parent class or classes in the case of multiple inheritances. This type of coupling is useful, but sometimes dangerous. It is dangerous because, through inheritance, an instance of a child class can be treated as though it were an instance of the parent class. It is useful, as it permits the development of significant software

Cohesion

IEEE [16] defines cohesion as "The manner and degree to which the tasks performed by a single software module are related to one another". Cohesion describes the "degree to which the elements of a portion of design contribute to the carrying out of a single, well-defined purpose" [33]. High cohesion at the class level signifies that all of the elements of the class are strongly related.

The lack of cohesion affects essentially the quality of a system. Testability decreases significantly, as the component you have to test carries no single, well-defined purpose and performs more than one functionality, which means it is harder to understand. You may either misunderstand its purpose or forget some of the purposes of the component. Also, having more than one purpose increases testing effort, as you have to trace more related components.

The lack of cohesion also decreases the reusability of these components, as they contain functions that are usually of no interest for the context in which they are going to be reused.

There are different types of cohesion defined in the literature. A significant one was again proposed by Timothy Budd. Budd states that the internal cohesion of a class is a measure of the degree of binding of the various elements within the structure. He identifies seven types of cohesion as a list, ranked on a scale from the weakest (least desirable) to the strongest (most desirable) as follows:

- Coincidental cohesion
- Logical cohesion
- Temporal cohesion
- Communications cohesion
- Sequential cohesion
- Functional cohesion
- Data cohesion

Coincidental cohesion occurs when elements of a class are grouped for no particular reason, often as a result of someone who tries to modularize a large program into several small units, by arbitrary segmentation. This type of cohesion usually indicates a poor design and existence of unrelated methods in a class.

Logical cohesion occurs when there exists a logical connection among the elements of the class, but none in either data or control. A typical example for such type of cohesion may be a package grouping due to functional reasons, such as mathematical functions (sine, cosine, etc.) assuming that there exists no references among these function classes.

Temporal cohesion occurs when elements are bound together because they all must be used at approximately the same time of execution. A typical example for such type of cohesion is a class that performs program initialization.

Communications cohesion occurs when methods of a class are grouped because they communicate with the same input/output data or devices. The class with such a cohesion acts as a manager class for the data or the device.

Sequential cohesion occurs when elements in a class are linked by the necessity to be activated in a particular order. This type of cohesion usually results from an attempt to avoid sequential coupling. Increasing the level of abstraction may help lowering this cohesion and obtaining a better design.

Functional cohesion occurs when the elements of a class all relate to the performance of a single function, which is a desirable type of binding. *Data cohesion* is the condition when a class defines a set of data values and exports routines that manipulate the data structure, as a class is used to implement a data abstraction.

Complexity and Size

IEEE [16] defines complexity as "(1) The degree to which a system or component has a design or implementation that is difficult to understand and verify; **(2)** Pertaining to any of a set of structure-based metrics that measure the attribute in (1)".

As the need for automation via software increases, software systems tend to become increasingly complex, day by day. This increase in size and complexity drastically affects several quality attributes, as well.

It is obvious that testability of a complex class requires much more effort with respect to a simple class. Complex components are also harder to understand and maintain, especially when the class to be tested is also low cohesive; incorporating more than one functionality. This effect makes classes more error-prone and consequently reduces their reliability. When you encounter a fault, it takes a substantial amount of time and effort to recover the source of the fault. Also, during maintenance, when any change to a part of the software is needed, it requires comprehension of the whole class.

E Da-wei classifies software complexity into four classes [10]. These four classes are:

- Domain Complexity
- Scale Complexity

- Artificial Complexity
- Functional Complexity

Domain complexity is directly created by the application domain or the problem space. Expertise help is necessary on the domain of the software you have to develop. Difficulty of communication among team members, especially developers and field experts may lead to product flaws, cost overruns, and schedule delays.

Scale complexity is induced by size or other scaling considerations. According to size perspective, software is considered to be one of the most complex forms of engineering. As it is very difficult to predict increase in complexity with the increase in size, due to the fact that the relation between size and scale complexity is nonlinear. Layering abstraction may reduce this type of complexity. Avoiding this complexity may also help performance increase in the software.

Artificial complexity is caused by the artifacts used for building software. This kind of complexity is generally caused by programmers, who try to change an often-repeated feature for an updated or new version of an application, as it is often difficult for programmers to find every instance of such a feature in millions of lines of code and possible for them to introduce new bugs. This kind of complexity comprehends structural complexity, programmer characteristics and problem complexity. It is hard to measure programmer characteristics objectively, while little work has been done to date on measures of problem complexity. Structural complexity instead has been studied extensively because it is the only component of psychological complexity, which can be assessed objectively.

Functional complexity helps us to investigate the work effort required to develop the software function, including decomposing and allocating the functional processes and designing each functional process to fulfill user needs as stated in the software specifications. It is not always possible to simplify complex core functions in an engineered system. An evolutionary process might be helpful and useful in such conditions to create an environment in which continuous innovation can occur.

Size is closely related to complexity, as the increase in size upraises complexity. In addition, a complex systems requires more code statements, and thus means an increase in size of the system. Software size and complexity are widely used to be able to estimate software development effort. To estimate effort, a connection has traditionally been made to the overall "size" of the system being developed, by means of an organization-specific value of team productivity. Size is one of the most obvious and easiest software factors to measure, but is not one of the best, as it is only available toward the end of the life cycle. A thorough analysis of the product metric domain suggests that "complexity" and not size may be more relevant to modern software systems.

Data Abstraction

IEEE defines [16] data abstraction as "(1) the process of extracting the essential characteristics of data by defining data types and their associated functional characteristics and disregarding representation details. (2) The result of the process in (1)".

Data abstraction may be defined as the essential characteristics of an entity that distinguish it from all other kinds of entities. An abstraction defines a boundary relative to the perspective of the viewer. Using abstraction allows selective information hiding based on scale issues. Classification is one particular form of abstraction, which means grouping of objects with similar or identical characteristics together in a common class.

Mitchell [29] defines three main goals of data abstraction as:

- Identifying the interface of the data structure. The interface of a data abstraction consists of the operations on the data structure and their arguments and return results.
- Providing *information hiding* by separating implementation decisions from parts of the program that use the data structure.
- Allowing the data structure to be used in many different ways by many different programs.

A system with a proper data abstraction displays a good-level of modularity, which makes it easily comprehensible. A class that represents an improper abstraction may either contain too many or no reasonable abstraction. If a class is too complex, it is very probable that it captures more than one abstraction. Such a class is probably not only excessively complex, but also non-cohesive. Thus, we observe that in this point the cohesion, complexity and abstraction good-design criteria converge [23].

Modularity

IEEE defines [16] modularity as "The degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components".

Modularity may be defined as the balance between low coupling and high cohesion is usually called modularity. If we consider cohesion at the module (subsystem) level, then a weak cohesion means that the system is not properly divided in subsystems, thus it has a lack of modularity.

Modularity is closely related to encapsulation and information hiding, which allow a modification to be made to the internal operations of an object by hiding the implementation details behind a public interface. Modifications may have side effects in other objects, in case modifications affect this public interface. It is expected to have a modular design so that changes to the internal operations of an object are contained within that object only.

One important way for programming languages to support modular programming methods is by helping programmers to keep track of the dependencies between different parts of a system. Interfaces and specifications are two important concepts in modular software development.

- *Interface*: A description of the parts of a component (a meaningful part of a program) those are visible to other program components.
- *Specification*: A description of the behavior of a component, as observable through its interface.

Having a modular design helps to reduce the costs associated with redesign and verification issues by allowing the programmer do this for every module independently. A module contains logical groups of classes and objects after applying abstraction and encapsulation processes. The whole groups of modules, each of which are connected among them, form the physical architecture of the software program. The Object-Oriented

languages make the distinction between the module's interface and its implementation, thus causing strict relation between encapsulation and modularization [23].

Encapsulation

IEEE defines [16] encapsulation as "A software development technique that consists of isolating a system function or a set of data and operations on those data within a module and providing precise specifications for the module".

Encapsulation may be defined as the process of splitting the elements that form the structure and behavior of an abstraction into individual compartments; encapsulation is used for separating the ”contractual” interface from its implementation.

The idea of encapsulation comes from two needs:

- The need to cleanly distinguish between the specification and the implementation of an operation,
- The need for modularity.

There are two views of encapsulation:

- The programming language view (the original view since the concept originated there),
- The database adaptation of that view.

The idea of encapsulation in programming languages comes from abstract data types. In this view, an object has an:

- *Interface* part
- *Implementation* part.

The interface part is the specification of the set of operations that can be performed on the object. It is the only visible part of the object. The implementation part has a data part and a procedural part. The data part is the representation or state of the object and the procedure part describes, in some programming language, the implementation of each operation.

Abstraction is the process that defines the object's interface and encapsulation defines the object's representation (structure) together with the interface implementation. The concealment of an object's structure and method implementation make up the so-called *information hiding* notion. [23]

The database translation of the principle is that an object encapsulates both program and data. In the database world, it is not clear whether the structural part of the type is or is not part of the interface (this depends on the system), while in the programming language world, the data structure is clearly part of the implementation and not of the interface.

Encapsulation provides a form of "logical data independence": we can change the implementation of a type without changing any of the programs using that type. Thus, the application programs are protected from implementation changes in the lower layers of the system.

It is a common belief that proper encapsulation is obtained when only the operations are visible and the data and the implementation of the operations are hidden in the objects. However, there are cases where encapsulation is not needed, and the use of the system can be significantly simplified if the system allows encapsulation to be violated under certain conditions. For example, with ad-hoc queries the need for encapsulation is reduced since issues such as maintainability are not important. Thus, an encapsulation mechanism must be provided by an Object Oriented Database Management System, but there appear to be cases where its enforcement is not appropriate.

Encapsulation separates the object interface from the object's representation so that one can modify the representation without affecting the various clients in any way because these depend on the server object's interface and not its implementation. It also allows the programmer to modify programs efficiently, with a limited and localized effort.

Inheritance

Budd defines inheritance as "the principle that knowledge of a more general category is also applicable to a more specific category". [8]

Inheritance allows; programmers to define classes incrementally by reusing previously defined classes as the basis for new objects, and classes to share their methods and fields. The set of methods and fields of our new class is composed of those defined by itself, and of those it inherits. The new class may override the methods it inherits, depending on the context of the object-oriented language.

Inheritance defines a relation among classes in which a class shares its structure and behavior with one or more other classes. A *child* class (or *subclass*) is the class that inherits attributes from a *parent* class, which ranks higher in the hierarchical tree. An *abstract parent* class is a class which is only used to create subclass and which does not have any direct instances.

In object-oriented environment, classes can be classified into a *hierarchical*
inheritance structure. This structure implies a hierarchy of the generalization/specialization type in which the class that derives specializes the more generalized the structure and behavior of the class from which it was derived. [23]

There are two kinds of inheritance in a *class hierarchy. Single* versus *multiple inheritance.* [15] Single inheritance means that a subclass is allowed to have only one single parent class. Multiple inheritance allows more than one single parent classes.

Beside *class hierarchy*, there also exists *object hierarchy*. From a semantic point of view, when class hierarchy indicates an ”*is a*” relationship, object hierarchy (or aggregation) indicates a ”*part of*” relationship. Aggregation defines a relationship between two objects where one of the objects is part of the other object.

For example, we may define two classes: apricot and fruit. Apricot is a fruit, which means there is an inheritance relationship between apricot and fruit classes, as apricot *is a* kind of fruit. A suitable example to object hierarchy may be wheel and car, as a wheel is a *part of* a car.

Inheritance heavily affects testability. The number of required test cases depends on usage of inheritance mechanism in the class and object hierarchy and the testing criterion of the project. As a class may inherit methods of other classes via inheritance mechanism, the testing criterion defines where to test these methods inherited into a class. Will they be in the inherited class, in the parent class or both? The answer will define the testing strategy and thus the required number of test cases. In case the testing criterion states to test all, both inherited and defined methods, in all classes, inherited methods will be test in both their own class and in the inherited classes, increasing number of test cases.

Polymorphism

The term polymorphic has Greek roots and roughly means “many forms”, as “poly” means “many” and “morphos” means “form”. Polymorphism allows the implementation of a given operation to be dependent on the object that contains the operation. To be able to better explain the concept, we give information on two other concepts.

The first of the two concepts is *interface.* The *interface* of a class is formed by the sum of all function signatures for the functions that can be called by clients of that particular object class. [23] In object-oriented environment, the objects are known inside the system only through their interfaces. An object’s interface does not give any information about its implementation. Therefore, it is possible that two different objects can implement the same interface in different ways.

The other of the two concepts is *binding.* Binding is the process by which a name or an expression is associated with an attribute, such as a variable and the type of the value the variable can hold. Depending on the moment when this binding takes place, there are two types of binding:

• Static binding (early binding) - the association is performed at compilation time.
• Dynamic binding (late binding) - the association is performed at run-time.

In dynamic binding, the request for an operation gets a correspondence only when the program is running. The possibility of substituting objects that have identical interfaces at run-time may be seen as the main advantage of dynamic binding. Using the concepts of binding and interfaces, we may also define polymorphism as the option of using some object in another object's stead when both objects share the same interface.

In the literature, there has been observed many different forms of polymorphism. The following four suggest by Budd seem to be the best of all. [8] These are:

- *Ad hoc polymorphism*, also known as *overloading*, defines a situation, where a single method name has several alternative implementations. The overloaded methods are distinguished at compile time based on their type signatures. All implementations have common method names, common output but different input variables. A typical example for this form of polymorphism is as follows:

```
public overloadedMethod (int input1 ){ . . . }
public overloadedMethod (int input1, String input2) { .
. . }
public  overloadedMethod  (int  input1,  String  input2,
double Input3) { . . . }
```

- *Inclusion polymorphism*, also known as *overriding*, defines a special form of overloading that appears within the context of the parent class/child class relationship. The two definitions have the same type signature, but one overrides the other one

```
class OverRiddenParent {
      public exampleMethod (int input1 ){ . . . }
}
class OverRidingChild {
      public exampleMethod (int input1 ){ . . . }
}
```

- *Assignment polymorphism*, also known as *polymorphism variable*, defines a variable that is declared as one type but in fact holds a value of different type.

```
Parent pClass = new Child();
// pClass declared as type Parent but hold type Child
```

- *Generics*, also known as *templates*, provide a way of implementing commonly-used tools and specializing them to specific situations. A generic class or function is parameterized by a type. By not specifying the type at the beginning, the function or class is allowed to be used in a wider range of situations. The following code section presents a sample of template, which implements a common function to obtain the maximum of two variables.

```
Template <class Temp> Temp max (Temp first, Temp last)
{
      if (first  <  last)
            return last;
      return first;
}
```

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Yurga, Tolga
Nationality: Turkish (TC)
Date and Place of Birth: 9 October 1977 , Manisa, TURKEY
Marital Status: Married
Phone: +90 532 567 40 54
email: tolgayurga@gmail.com

EDUCATION

Degree	**Institution**	**Year of Graduation**
MS	Bilkent University, M.B.A.	2001
BS	Bilkent University, Electrical & Electronics Engineering	1999
High School	Fatih Anadolu High School, Manisa	1995

WORK EXPERIENCE

Year	**Place**	**Enrollment**
2006- Present	Todem Bilişim Danışmanlık	General Manager, Partner
2001-2006	Emek Bilişim	R&D Manager
1999-2001	MilSoft Software Tech.	Systems & Software Engineer

FOREIGN LANGUAGES
Fluent English, Beginner German

PUBLICATIONS

1. Yurga T., " A New Model To Measure Testability in Object Oriented Software Design, National Software Engineering Symposium (UYMS'2005), 5(12), 225-228 (2005)

2. Yurga T., Dogru H. A. "A New Model To Assess The Testing Process And Testability Of Object-Oriented Software Systems", Wiley InterScience - Software Testing, Verification and Reliability (2009) *(Submitted - Being Processed and Reviewed)*

HOBBIES

Movies and TV Shows, Photography, Electronics, Cars