Object Oriented Software Quality Estimation Using Maintainability Metric and Genetic Algorithms

THESIS

Submitted in partial fulfilment
of the requirements for the degree of
DOCTOR OF PHILOSOPHY

By

Nirmal Kumar Gupta

Under the Supervision of

Prof. Mukesh Kumar Rohil

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
2014

Object Oriented Software Quality Estimation Using Maintainability Metric and Genetic Algorithms

THESIS

Submitted in partial fulfilment
of the requirements for the degree of
DOCTOR OF PHILOSOPHY

By

Nirmal Kumar Gupta

Under the Supervision of

Prof. Mukesh Kumar Rohil

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
2014

I wish to dedicate this thesis to My Family
for their continued support
and encouragement

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI (RAJASTHAN) INDIA

C E R T I F I C A T E

This is to certify that the thesis entitled "**Object Oriented Software Quality Estimation Using Maintainability Metric and Genetic Algorithms**" and submitted by **Nirmal Kumar Gupta** ID. No. **2005PHXF424P** for award of Ph.D. Degree of the Institute, embodies original work done by him under my supervision.

(Signature in full of the supervisor)
PROF. MUKESH KUMAR ROHIL

Associate Professor
Computer Science and Information Systems Department
Work Integrated Learning Programmes Division
Birla Institute ofTechnology and Science-Pilani
Pilani – 333 031 (Rajasthan) INDIA

Date:

Acknowledgements

First of all, I wish to thank my supervisor Prof. Mukesh Kumar Rohil, Associate Professor, Computer Science and Information Systems Department, Birla Institute of Technology and Science, Pilani, Rajasthan, India, for his valuable guidance, encouragement and moral support. It has been a great pleasure to be associated with him on this work.

Special thanks are due to Prof. B. N. Jain, Vice Chancellor; Prof. G.Raghurama, Director; Prof. Ranendra N. Saha, Dean, Sponsored Research and Consulting Division; Prof. S K Verma, Dean, Academic Research Division, Birla Institute of Technology and Science, Pilani (BITS Pilani), (Raj.), for giving me an opportunity and encouraging research in the Institute. Further, I sincerely acknowledge the encouragement and help received from Prof. J. P. Mishra, Unit Chief, Information Processing Unit, Prof. Navneet Goyal, Head of Department and Chairman, Doctoral Research Committe, Prof. Sudeept Mohan, Convernor, Doctoral Research Committe, Department of Computer Science & Information Systems, BITS-Pilani, at various stages of the work. Further, I sincerely thank member of the doctoral advisory committee, Prof. Sundar Balasubramaniam and Dr. Yashvardhan Sharma for their suggestions and for helping me in improving the work. I also thank my former supervisor, Dr. Dinesh Kumar Saini, Associate Professor, Sohar University Oman for his support and advice during preliminary stage of the research.

Sincere thanks are also due to Dr. Virendra Singh Nirban, Assistant Professor, Publications and Media Relations Unit, BITS Pilani, Further, I sincerely thank all the persons who directly or indirectly helped me at various stages of the work. I extend my thanks to all the faculty members of the Department of Computer Science & Information Systems, BITS Pilani for their suggestions during the departmental research seminars.

Finally, but most deeply, the author thanks his parents, his wife, his daughter, his son and other family members for their love and moral support during the entire period of this research work with which this work is successfully completed.

(Nirmal Kumar Gupta)

Abstract

The software developers are facing a major challenge that over 70% of the software development effort is spent in testing and maintenance of software. Software testing is the most common software quality assessment technique. Software quality cannot be added to the software by testing it, instead software must be developed in a way that guarantees that the software has high quality in every phase. High quality software can be assured by applying the appropriate measurement and testing techniques during software development. Quality Attributes such as maintainability, reusability & testability are useful to find out the extent to which software is useful to undergo changes during the usage phase. At unit testing level, measuring how well a software class can be reused and maintained helps programmers to write reusable and maintainable software, and also helps to identify reusableormaintainable class components. It is widely agreed that there is a direct relationship between poor maintainability and high coupling. In software design there can be various kinds of connections which comprise a coupling relationship. One of such hidden connections between any two seemingly unrelated parts of the system gives rise to indirect form of coupling.

To investigate the relationship between indirect coupling and maintenance effort of the object oriented software, this research proposes a metric called Indirect Path Coupling that measures coupling of a class through indirect coupling paths to other classes in the system. The proposed metric takes into account all the indirect coupling paths which are formed between any two classes in the software by considering independent, multiple or partially overlapping multiple indirect coupling paths into account. Extensive case studies have been conducted on several releases of nine open source software to provide empirical software maintenance effort. Indirect path coupling for various classes has been computed by taking into account the various indirect paths formed of different lengths. It is found that the indirect path coupling is correlated to maintenance effort at a statistically significant level and the proposed metric can be used as a measure of maintenance effort in software.

Another aspect of software quality is its testing using some test criteria. Software testing is an expensive process, therefore automated Test Data generation has become vital in Software Testing process. Automating the generation of object-oriented unit test-cases for structural testing techniques has been challenging many researchers because of the various factors like cost saving and improvement in test quality. Testing process requires test sequences to be generated, each of which models a particular scenario in which the class under test is examined. Basically two kinds of input test data are required for

testing object oriented software. The first kind of data has to produce the right sequences of method calls, and the second kind is required to bring the object under test in the required state for testing. To handle the state problem of Object-Oriented programs it requires the development of carefully fine-tuned methodologies that could promote the transversal of problematic structures and difficult control-flow paths.

To promote the transversal of such problematic structures and difficult control-flow paths, the presented research proposes techniques for generating test cases using a model based on finite state machine specification and by applying genetic algorithms. The fitness of test cases has been evaluated using genetic algorithms leading to the improvement of search process by achieving higher coverage and evolving more number of infeasible test cases into feasible ones. Quality of both feasible and infeasible test cases are considered and the proposed technique helps to improve those infeasible test cases which have better possibility to be developed into feasible ones at certain stages, promoting diversity and enhancing the possibility of achieving full coverage.

An algorithmhas been presented for testing an object oriented software class based on genetic algorithms. It uses potential solutions to encode as tree structure genetic individuals which facilitate to apply mutation and crossover operators over them. This representation is particularly suited to represent and evolve object oriented programs which can be represented as method call trees. In order for a test sequence to be executed, the method call tree must be decoded into test program. Test program fitness evaluation involves the instrumented method under test in order to collect information about program behavior during execution. The structural testing involves Control Flow Graph (CFG) analysis of this instrumented code. The proposed approach introduces some parameters such as hit-count factor, path factor, weight factor etc. to compute the fitness of feasible as well as infeasible test cases. This approach works more efficiently with maximum improvement once the defined parameters are selected accurately.

The proposed technique has been validated through various case studies over triangle classification program and on various other classes from util-package of standard java library. The results of the case studies show that by selecting the defined parameters appropriately one can maximize the improvement of feasible as well as infeasible test cases with maximum coverage. For triangle classification program a highest value of improvement factor 2.16 is obtained for feasible test cases and 0.92 is obtained for infeasible test cases. Similarly for various other classes of the util package a highest value of improvement factor 1.57 is obtained for feasible test cases and 0.88 is obtained for infeasible test cases.

Table of Contents

List of Mathematical Notations

Notations	Details
$D(M)$	The domain D over set of methods $M$for the genes, defined as maximum number of test cluster methods.
S_C	Ordered set of constructors $(c_1,c_2,....c_r)$
S_M	Ordered set of methods $(m_1,m_2,....m_n)$
$\mathbb{N}$	Set of all natural numbers.
$D(T_i)$	The domain D of target genes for some statements$_i$
O_t^i	Ordered set of objects which are instances of class t and have been created by the statements which are called before statements$_i$
$D(R_i)$	Domain of receiver objects defined as the objects created by method call in statement s_i.
m_k	kth method in S_M
G_j	Gene encoded for method m_j
ρ	Function which assigns each value of gene to corresponding target object.
ξ	Function which assigns each value of gene to a primitive value or object reference.
N_{ni}^h	Hit Count Factor
α	Weight Factor
H_t	Set of nodes which are traversed by the test-case t.
β	Infeasibility Factor
H_d^{ne}	Set of all nodes which are descendants of node n_e in CFG.
IF_F	Improvement Factor for feasible test cases.
IF_I	Improvement Factor for infeasible test cases.

List of Abbreviations

Abbreviation	Details or Expanded Form
CBO	Coupling Between Objects
CC	Cyclomatic complexity
CCC	Complexity, Coupling, and Cohesion
CCM	Class Coupling Metrics
CFG	Control Flow Graph
CK metrics	Chidamber & Kemerer Metrics
CMMI	Capability Maturity Model Integration
CR	Combination Rate
CTC	Composite Transitive Coupling
CUT	Class Under Test
DAC	Data Abstraction Coupling
DIT	Depth of Inheritance Tree
EA	Evolutionary Algorithm
ES	Evolution Strategies
ET	Evolutionary Testing
GA	Genetic Algorithm
GP	Genetic Programming
HCF	Hit Count Factor
IEEE	Institute of Electrical and Electronics Engineers
ICAMA	Indirect Coupling and Maintenance Analyzer
IF	Improvement Factor
IPC	Indirect Path Coupling
ISO	International Organization for Standardization
LCOM	Lack of Cohesion of Methods
LoC	Lines of Code
MCS	Method Call Sequence
MOOD	Metrics for Object Oriented Design
MPC	Message Passing Coupling
MR	Mutation Rate
MUT	Method Under Test
NUCD	Number of used classes by dependency relation
OO	Object Oriented

Abbreviation	Details or Expanded Form
PF	Path Factor
QMOOD	Quality Model for Object Oriented Design
RNUCD	Ratio of NUCD
SDLC	Software Deveelopment Life Cycle
SE	Software Engineering
SRCC	Spearman Rank Correlation Coefficient
SQA	Software Quality Assurance
STC	Simple Transitive Coupling
STGP	Strongly Typed Genetic Programming
SUT	Software Under Test
UML	Unified Modeling Language
WF	Weight Factor

List of Figures

List of Tables

CHAPTER 1 *Introduction*

Today, software plays an important role because a substantial proportion of all products, both commercial products and other application domains, contain some kind of software. And the trend is that each product contains more and more software features every year. One major factor, independent of the field of application of the software, is the quality of the software product. If the product quality is below expectations the customers will shortly find a substitute product that better suits his or her needs. Therefore, software development organizations are forced to ensure that their products are of acceptable quality and not exceeding the budget of the customer. There is no standard definition for good software quality in industry but it is considered as the value (satisfaction) it provides to the users, mekes profit for developers, its capability to generate as few as possible number of complaints and in some way its contribution to the goals of humanity [Boehm 2003].

Software metrics are proposed as potential tool in the endeavor to improve the quality of computer software.In general, such requirements for good quality needs to be satisfied by the people involved in the development and support of these software systems through various Software Quality Assurance (SQA) activities, and the claims for good quality need to be supported by evidence based on concrete measurements and analyses.

SQA consists of a means of monitoring the software engineering processes and methods used to ensure quality. The methods by which this is accomplished are many and varied, and may include ensuring conformance to one or more standards, such as ISO 9126 or CMMI. One of the most important parts of SQA and most commonly performed SQA activities is software testing. Software testing involves the execution of software and the observation of the program behavior or outcome. Software testing by itself does not improve software quality. Software test results are an indicator of quality, but in and of themselves, they don’t improve it.

The presented research addresses the issue of improving software quality. Software organizations use Software Quality Assurance activities to ensure producing quality software. Ensuring quality software has two aspects, first how to measure quality in software and second, how to test software

which can help in improving software quality. In this research the work is broadly divided in two parts, the first part investigates the metrics which can predict some aspect of software quality in software development process which may help to build quality software. The second part investigates the techniques which automate the process of test data generation. The generated data can be used to test the software and build confidence in it.

1.1 Characteristics of Software Quality

Software quality can be divided into internal and external quality characteristics. External quality characteristics are those that the users are aware of, including the following [Lincke and Lowe 2006]:

- **Correctness** Absence of defects in specification, design, and implementation of a system.
- **Accuracy** Absence of errors in a system, especially with respect to quantitative outputs. Accuracy differs from correctness; Accuracy determines how well system does the job it was built for, correctness determines whether it was built correctly.
- **Usability** Ease of use and low learning curve of a system.
- **Efficiency**Optimised use of system resources, including memory and execution time.
- **Reliability** The ability of a system to perform its required functionality at all times.
- **Integrity** Proper protection of a system's programs and data from unauthorized access.
- **Adaptability** System's ability to perform, without modifications, in environments that it was not designed for.
- **Robustness** System's ability to continue performing with invalid inputs and in stressful conditions.

Some of these external quality characteristics overlap and some conflict with each other, e.g. having a highly efficient system usually means that the system is not very adaptable. Internal quality characteristics are those that the users are not aware of. However, developers constantly deal with internal quality characteristics and care about them. Internal quality characteristics include.

- **Maintainability** Ease of changing the system, adding capabilities, improving performance, or correcting defects.
- **Flexibility**The ease with which a software component or software system can be modified for use in some applications or in an environments which is not specifically designed for it.

- **Portability**The degree to which software running on one platform can easily be converted to run on another platform.
- **Reusability** Possibility and easiness of using parts of the system in other systems.
- **Readability** Ease of reading and understanding low-level source code of the system.
- **Testability**Ability of software system to be tested on different levels, especially in unit testing and system testing.
- **Understandability** Ease of comprehending the system at both low-level and high-level.

Users care only about the external quality characteristics, but external quality characteristics often result from the internal quality characteristics. In order to have a high quality product, internal quality characteristics must be in a good shape, otherwise the external quality will slowly but surely deteriorate [Lincke and Lowe 2006].

1.2 Software Quality, Testing and Metrics

There are five major views (transcendental, user, manufacturing, product, and value-based views) of software quality [Garvin 1984]:

1. In the transcendental view, quality is hard to define or describe in abstract terms, but can be recognized if it is present. It is generally associated with some intangible properties (ease of use, easy to install, less number of crashes etc. from user's point of view) that delight users.
2. In the user view, quality is fitness for purpose or meeting user's needs.
3. In the manufacturing view, quality means conformance to process standards.
4. In the product view, the focus is on inherent characteristics in the product itself in the hope that controlling these internal quality indicators (or the so-called product-internal metrics) will result in improved external product behavior (quality in use).
5. In the value-based view, quality is the customers' willingness to pay for software.

Different standards organizations have different definition for their quality standards. IEEE and ISO are the most widely used standards for computer science. There is a standard quality model, called ISO 9126 [ISO 2001]. In this standard, the quality of software is defined to be:

> *The totality of features and characteristics of a software product that bear on its ability to satisfy stated or implied needs.*

A significant role is played by the software testing achieving and assessing the quality of a software product [Osterweil 1996]. Generally organizations improve the quality of the products as they repeat a *test–find defects–fix* cycle during development also they assess how good the software system is, by performing system level tests before releasing a product. Therefore, Friedman and Voas [1995] have described software testing as a verification process for assessment of software quality and its improvement.

For determing the software faults, the software testing has been proved to be an important and valuable activity performed during development cycle. Three levels of software testing are defined: namely unit/component level testing, integration testing and system level testing [Myers 1979]. The objective of testing at each of these three levels is different. At unit/component level the aim is to show whether the unit/component satisfies its functional specification and/or whether its implemented structure matches the intended design structure. In integration testing the aim is to show whether there are inconsistencies between units or components.At System level the main concern is about those issues and behaviors that can only be exposed by testing the entire integrated system or a major part of it [Binder 1996]. Also there are two general testing approaches called as white-box and black-box testing. Out of these two approaches the white box testing approaches, which involves branch testing and path testing; requires the implementation knowledge of the source code. On the other hand, Black-box approaches, which involve functional testing and random testing, require knowledge of the specification details. These two approaches are complementary to each other [Mathur 2008].

1.2.1 Types of Software Testing

In general, two types of software testing methodology are adopted (i) Execution-based testing (dynamic testing) and (ii) non execution-based testing (static testing).In Execution-based testing the software must actually be compiled and run while non execution-based testing involves examination of the program's code and its associated documentation but the software isn't actually executed. The latter is basically the verification portion of Verification and Validation while former is the validation portion of it. Figure 1.1 shows broadly the different types of software testing techniques [Binder 1996].

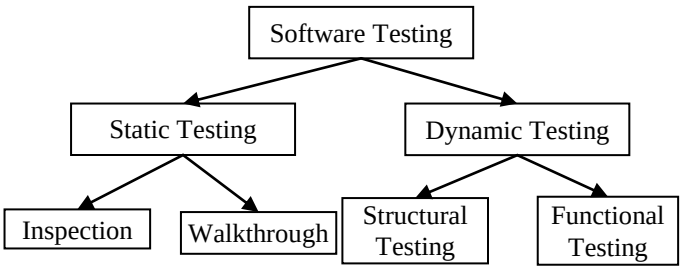


Figure 1.1 Classification of software testing

1.2.1.1 Functional Testing

This is a dynamic test method based on specificationsof the software. This is also called as Black Box Testing, because it assumes no knowledge of how the system or component is structured inside the box.In essence, the tester is concerned with what the software does, not how it does [Mathur 2008].

1.2.1.2 Structural Testing

Structural testing uses the internal structure of the software to derive test cases. It is also commonly called Glass Box or White Box technique. Since it requires knowledge of how the software is implemented, structural testing considers the program code, and test cases are designed based on the logic of the program so that every element of the logic is covered. [Pressman 2005]

1.2.2 Software test automation

When software testing is done manually by a person who primarily creates and executes test cases, then the results of test execution are compared with the expected output and if any defects are detected then such defects are logged. All the above process is performed manually. There are several manual testing methods exist in literature [Jorgensen 2002].

The activities like executing the test cases, comparison of results and logging of defects are performed automatically without human intervention in automated software testing. In recent time the automated software testing has received a lot of attention as a means to reduce testing costs, find more number of defects, and save valuable development time. In simpler words it can be said as “writing code to test code”. Specialized testing tools are used to perform most of the test automation

process. But the use of automated tools requires more specialized skills than those needed for manual testing.

1.2.3 Quality of Object-Oriented Class Unit

Unit testing is the process of testing the individual units (functions, classes) of a program in isolation in order to find faults. The most basic way to write unit tests is to write a set of simple methods or procedures which exercises the unit under test. These tests can then be manually executed to ensure that the code is properly tested. Although this is perfectly valid way to do unit testing, frameworks exist for most languages which can help automate the process of running unit tests, and provide some of the boiler-plate code that the tester has to write. In addition a framework can also automatically setup the environment in which the test is run to ensure that each test is run in isolation. Units that are tested can be single functions, modules, or combination of modules. Unit tests are done in isolation. Most common way to do unit testing is to automate the tests. Automated unit tests are tests that run a piece of code, check the result, compare it to the expected and report the status of the tests.

Defects found using unit testing is usually easy to fix, and when developers fix their own defects, the rest of the software is not affected by these defects. Usually unit testing finds defects in the external quality characteristics, but it can also find defects in the internal quality characteristics, especially in testability. Depending on the framework used, unit testing can be very easy to use, but if there is not any framework available, it could take too much effort to write and execute unit tests.

1.2.4 Software Quality and Metrics

ISO 9126 [ISO 2001] provides a hierarchical framework for quality definition, organized into quality characteristics and sub-characteristics. There are six top-level quality characteristics, with each associated with its own exclusive (non-overlapping) sub characteristics:

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability

- Portability

There are broadly two different sets for the product quality characteristics mentioned above. These two sets are: external and internal. External quality characteristics such as: functionality, reliability, usability, efficiency, flexibility, friendliness, simplicity etc. are of more concern from a customer's point of view. The reason for this is because these are the characteristics which are easily explored by use of the product.

On the other hand, software developers are more concerned about the internal quality characteristics such as: maintainability, reusability, portability, testability, etc. since these are the characteristics which are related the software development effort.

Most of the quality characteristics discussed above cannot be measured before the system is used for a certain period of time. However, there exist some software attributes that can be measured during the software development cycle, and which can be used as indicators of them. Examples include cohesion and coupling. To illustrate how certain software quality attributes can be indicators of some quality characteristics, consider the example of maintainability of a software component (a class in an object-oriented system, for example). Assessing the maintainability of a software product is very important because it helps produce high quality software more quickly [Basili et al. 1996]. It is not possible to directly measure maintainability. However, the coupling (software components depend on each other for completion of a task) of a software component can be a good indicator of its maintainability. Coupling is a directly measurable quality, maintainability is not.

In order to determine quality of software we must have some metrics to measure quality. Measurement enables the organization to improve the software process; assist in planning, tracking and controlling the software project and assess the quality of the software thus produced. By controlling, it is meant that one can assess the status of the process, observe trends to predict what is likely to happen and take corrective actions for modifying our practices. Measurement is an essential element of management; there is little chance of controlling what we can not measure.

Therefore organizations developing software products, need some measurement techniques to check the extent to which the product satisfies the above specified quality characteristic, or the characteristics specified by some other standard. This is when the metrics comes in to existence. Its purpose is to make assessments during and after the software development, so as to know whether the software quality requirements are being met or not.

1.3 Metrics for Object-Oriented Programs

Object oriented software development methodology is becoming more pervasive, therefore it is expected from software engineers to use quantitative measurements for software design quality assessment at both the architectural as well as components level. The software designers may be able to use these measures to access the software early in the process, which will make the changes to reduce complexity and improve the continuing capability of the product. Object-oriented design exhibits four main features: inheritance, data abstraction, dynamic binding, and information hiding [Lewis and Wiener 1998]. These features allow software developers to cope with the complexity of large software and develop a manageable software design. But to achieve expected benefits and advantages of such inherent features of object-oriented methodology, it is necessary to establish some basic standards and guiding principles that the developer should follow. This methodology may be used in measurement of the metrics for object-oriented software. There exist a number of design methodologies that suggest the guiding principle for various ways for the development of object-oriented software system. The measurement process is to drive the software measures and metrics that are appropriate for the representation of software that is being measured [O'Regan 2002]. Software metrics deal with the measurement of the software product and the development process.Software metrics provide measurement for certain aspects of software. The usage of metrics will reduce the subjectivity during the assessment of software quality and it provides quantitative basis for making decisions about the software quality.

As the development of object-oriented software is rising, researchrs in this field are introducing more and more metrics for object-oriented languages. Various metrics have been proposed related to various object-oriented constructs like class, coupling, cohesion, inheritance, information hiding and polymorphism. Some of the important methods of evaluating an object-oriented design quality can be through the use of measures for coupling and cohesion.

Software metrics evaluate different aspects of the complexity of a software product. Software complexity was originally defined as “a measurement of the resources that must be expended in developing, testing, debugging, maintenance, user training, operation, and correction of software products” [Shooman 1983]. Complexity cannot be directly avoided in design. There are many factors that contribute to the complexity. The complexity influences some of the wanted quality characteristics mentioned in section 1.2.4.

1.3.1 Software Complexity and Maintainability

Software complexity is assumed to be a multi dimensional construct [Henry and Wake 1991]. The complexity of a program depends upon its magnitude, the complexity of its control structure, and the complexity of its data flows [Basili and Hutchens 1983]. Other researchers add other factors to this list, such as the degree of modularity [Bowen 1978]. Munson and Khoshgoftaar [1989] conclude that four or five such complexity factors suffice to describe the multi-dimensional complexity of a program.

Basili [Basili et al. 1996] defines software complexity as *"...a measure of the resources expended by another system whileinteracting with a piece of software. If the interacting system is people, the measures are concerned withhuman efforts to comprehend, to maintain, to change, to test, etc..that software.*". Complexity in software code tends to increase the human efforts used to comprehend, maintain, change or to test the software code. Factors that increase maintenance effort will increase the overall cost of the software, since maintenance costs are most directly a function of the professional labor component of maintenance activities.

Maintainability is one key external attribute that significantly affects software development. In today’s industry it is expected for a system to change continuously to accommodate new features or to adapt to changing requirements. Maintenance is in fact one of the biggest factors of cost, with figures reported to be around 60 to 80% [Erdil 2003]. Thus it is important to look into what makes maintenance so difficult. To understand the difference between maintenance and maintainability we consider the following definitions:

Software maintenance is defined as "*the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment*" [IEEE 2006].

Software maintainability is defined as "*the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment*" [IEEE 2006].

It is clear from the above definitions that the maintenance is the process performed as part of the Software Development Life Cycle (SDLC) whereas maintainability is the quality attribute associated with the software product. These are two inherently different but interrelated concepts. The cost of a software system is influenced by the maintainability of a software system. Therefore, it is important to be able to forecast a software system's maintainability so that the costs could be managed effectively.

1.3.2 Coupling Metrics

Coupling was first introduced in the context of structured development techniques [Stevens et al. 1974]. Here coupling is defined as "the measure of the strength of association established by a connection from one module to another". High coupling between classes is considered as bad design while low coupling between classes is advocated by Pfleeger and Atlee [2006]. Strong coupling complicates a system since a module is harder to understand, change or correct, if it is interrelated with other modules. A good design of coupling between classes allows classes to change its functionality during maintenance without affecting the other classes it is coupled with. Therefore, ideally it should be modified without taking the change of other classes into account.

Classes or objects may be coupled three ways:

- The objects are called to be coupled, whenever a message is passed from one object to another.
- There exists a coupling between classes, whenever methods declared in one class use methods or attributes from the other classes.
- A strong coupling is shown between super classes and their subclasses because of inheritance.

Coad and Yourdon [1991] extended the principle of low coupling to object-oriented software. Coupling Between Objects (CBO) and Cyclomatic Complexity have long been used to measure software quality and predict maintainability and reliability of software systems prior to release. CBO is a count of the number of other classes to which a class is coupled. The measurement of CBO is performed by counting the number of distinct non-inheritance related class hierarchies on which a class depends. The more number of couplings is undesirable as it is opposite to the philosophy of modular design and prevents reuse [Kearney et al. 1986]. In particular, CBO has been shown to correlate with fault-proneness and maintainability of a system at the class level. However, the CBO metric is based on a static analysis of class code, and the ability of the CBO metric to accurately predict the actual amount of coupling between objects is as yet unproven.

1.3.3 Cohesion Metrics

The cohesion of a module is the extent to which its individual components are needed to perform the same task [Fenton and Pfleeger 1998]. In object oriented paradigm of software development, class cohesion can be thought as "the measurement of relatedness among the members of class" [Bieman and Kang 1995]. The word "relatedness" in object oriented context means the similarity in the methods exposed by a class implementation. High cohesion indicates good class subdivision. There are seven categories of cohesion which range from the most desirable (functional) to least the desirable (coincidental) [Yourdon and Constantine 1979]. It is possible for a class to exhibit more than one type of cohesion. It is desirable that a good software design must have highly cohesive classes i.e. preferably functional. Effective object-oriented designs maximize cohesion since it promotes encapsulation.

The software complexity may increase because of lack of cohesion or low cohesion; therefore it increases the likelihood of errors during development. Software classes having lower value of cohesion could possibly be subdivided into two or more subclasses with increased cohesion. Efficiency and Reusability are evaluated by using this metric [Sharble et al. 1993; Hudli et al. 1994; Lorenz and Kidd 1994]. The existing class cohesion metrics can be categorized into two types, namely implementation metrics and design metrics. The implementation metrics are those metrics

which are computed from the source code, whereas the design metrics are computed from the design of a system.

There are at least two ways to measure cohesion:

- For each data field in a class, calculate the percentage of methods that use that data field. Average the percentages, then subtract from 100 percent. Lower percentages indicate greater data and method cohesion within the class [Rosenberg and Hyatt 1997].
- Methods are more similar if they operate on the same attributes. Count the disjoint sets produced from the intersection of the sets of attributes used by the methods [Rosenberg and Hyatt 1997].

In CK metrics suite [Chidamber and Kemerer 1991] Lack of Cohesion of Methods (LCOM) metric was used for class cohesion. A high value of cohesion indicates good class subdivision. The complexity of software design increases because of Lack of cohesion or low cohesion, therefore it increases the possibility of errors during the software development process.

1.4 Class Level Test Data Generation

Class is considered as the basic unit in object oriented software. Testing can be done at four different levels of abstraction found in object oriented software. These are the method level, class level, cluster level, and system level. The method level considers the code for each operation in a class. The class level is composed of the interactions of methods and data that are encapsulated within a given class. The cluster level consists of the interactions among cooperating classes, which are grouped to accomplish some tasks. The system level is composed of all the clusters [Smith and Robson 1992].

Object-oriented programs contain various features which require specific attention while performing software testing: the hidden state of objects, inheritance, polymorphism, dynamic binding, and exception handling. The test cases generated must be complemented with test cases generated using approaches which are specifically designed for addressing various object-oriented features. The different test suites generated for dealing with different characteristics of the application under test is generally good practice. Inheritance must be considered carefully and it allows reducing the testing effort by avoiding retesting those methods which are already tested in the ancestor classes. If the

inheritance is simply ignored by flattening class hierarchies it may simply result in the generation of redundant test cases. Specific techniques, such as the testing history approach presented in [Harrold and McGregor 1992], can be used to reduce the number of test cases to be generated.

Testing object-oriented programs in presence of polymorphism and dynamic binding requires considering different bindings for each polymorphic call. Such bindings can be computed with specialized techniques (e.g., the technique proposed in [Orso and Pezze 1999]). Additional test-case specifications could be generated by the presence of information about bindings needed to test. Software testing is complicated by exception-handling constructs due to the presence of implicit transfers of control that occur when exceptions are raised. Researchers have introduced the testing techniques which are specifically designed for addressing exception handling (e.g., the technique proposed in [Sinha and Harrold 1999]).

1.5 State-of-the-Art of Software Maintainability Metrics

One of the earliest software product quality models was suggested by McCall [McCall et al. 1977] and his colleagues. McCall's quality model defines software-product qualities as a hierarchy of factors, criteria, and metrics and was the first of several models of the same form. International efforts have also led to the development of a standard for software-product quality measurement, ISO 9126. It defines the following six characteristics: functionality, reliability, usability, efficiency, maintainability and portability.

In order to better quantify the concept of quality, researchers have developed various indirect models that attempt to measure software product quality by using a set of quality attributes, characteristics, and metrics. While defining these quality models, an important assumption is that the internal product characteristics (internal quality indicators) influence external product attributes (quality in use), and by evaluating a product's internal characteristics some reasonable conclusions can be drawn about the product's external quality attributes. This product-based approach is frequently adopted by software-metrics advocates because it offers an objective and context independent view of quality [Kitchenham and Pfleeger 1996; Bansiya 2002].

Maintainability is an important characteristic of object oriented software. Maintainability can be measured by considering two broad approaches, which are reflecting external and internal views of the attribute. Maintainability is an important quality factor, and it depends not only on the product itself but also on the programmer performing the maintenance. Maintainability can be measured by using a external and more direct approach which requires to define measures of the maintenance process and then collect the opinion of the programmers who participate in this process. But the problem with this approach is that it is time consuming and depends on conducting a survey, which may be of a high cost. Also such kind of surveys generally requires the improvement of the accuracy and the interpretation of their input and their derived results [Rosenberg and Hyatt 1997]. Another alternative internal approach using the internal metrics and expect that these are metrics predict the programmers' opinion of the maintainability of a software product. Another advantage is that, in this approach, the measures can be gathered earlier and easier. Generally it happens that while performing the software maintenance process the structure of the system usually degrades. Random patches applied by members of the maintenance department often result in a low quality system structure. It becomes more difficult to maintain the system with time. The degradation of a system can be controlled by software metrics in order to keep the quality of the maintenance process at a high level.

The maintenance effort for a software application depends on measurable metrics that can be derived from the software development process. The maintainability index [Oman and Hagemeister 1994] which determines the maintainability of software system and which is based upon the status of the source code shows high correlation between assessment automated models and some expert evaluation. The data of maintenance is collected by Binkley and Schach for the development of project written in any language like C, C++, COBOL etc and produce a level of interaction between modules, having low coupling were subjected for less number of maintenance effort and fewer maintenance fault and failures [Binkley and Schach 1998].

Genero et al. [2003] proposed some internal and some external attributes which can be used in early development of Object-Oriented (OO) software to analyze structural complexity and size of UML. They also serve as maintainability indicators and can be used to gather empirical data to turn in the basis of current study and also define some measuring properties of object-oriented software such as

inheritance, cohesion and coupling [Genero et al. 2003]. Hayes et al. [2004] proposed the adaptive maintenance effort model which can be used to determine the line of code change and also describe the regression model for adaptive maintenance, which can provide the useful information for manager and maintainer.

The way in which various software components are related provides the key for various quality attributes. Coupling is such a metric which can be used to estimate various quality attributes such as maintainability, complexity, and reliability. One of the important features of object oriented languages is that they are designed to minimize the dependencies between classes which further reduce the coupling [Offutt et al. 2008].

CBO is an Object-Oriented metric that measures the different coupling relationship between objects. This original metric assigns a measurement of one to each coupling relationship. However, when CBO is paired with the Cyclomatic Complexity (CC) it can help measure the weight of a coupling connection. These connections modify the strength of the relationship and therefore must be taken into account when defining the coupling complexity between objects.

In recent years, many new coupling metrics have been introduced - with and without empirical validation [Arisholm 2002; Gui and Scott 2006]. The goal is to fuse two metrics together; CC and an Object-Oriented metric, CBO, and propose novel metrics which combine the two to measure class quality (defect or error-proneness of a class).

1.6 State-of-the-Art of Class Level Test Case Generation

Class is the basic unit of testing in object oriented software. Testing of the class involves testing of the methods defined in the class and their impact on the state of the class/objects. Different approaches are used to test the methods of the class. In the effort to improve the existing testing infrastructure, a number of techniques have been developed to automate the test execution; however, the automation of test data generation is still a topic under research. Recently, a number of methods such as metaheuristic search, random test generation and static analysis have been used to completely automate the testing process, but the application of these techniques to real software is still limited. Random test case generation has been used to automate the generation of test cases, but a number of

studies found a genetic algorithm (evolutionary testing) to be more efficient and to outperform random testing [Watkins 1995; Michael and McGraw 1998] for structural testing of the software.

The study of genetic algorithms as a technique for automating the process of test case generation is often referred to as evolutionary testing in the literature. Since the early 1990s, a number of studies have been done on evolutionary testing [Baresel et al. 2002; Tonella 2004; Harman and McMinn 2007]. The complexity and applicability of these studies vary. In order to classify the relevance of past research for this project, a number of studies have been classified according to the complexity of the test cases being generated and the optimization parameter used by the genetic algorithm. The complexity of the test cases being generated is important because to generate test cases for structured programs that only take simple input, such as numbers, is simpler than generating test cases for object-oriented programs, which is one of the goals of this research.

Branch coverage was the most common optimization parameter used to drive the evolution of test cases. However, there is little evidence of a correlation between branch coverage and the number of uncovered faults. Although code coverage is a useful test suite measure, the number of faults a test suite unveils is a more important measure. Past research has shown that evolutionary testing is a good approach to automate the generation of test cases for structured programs [Watkins 1995; Michael and McGraw 1998; Wappler and Lammermann 2005]. To make this approach attractive to industry, however, the system must be able to automatically generate test cases for object-oriented programs and to use the number of faults found as the main optimization parameter.

One of the problems when evolving test cases for object-oriented programs is the initialization of an object into a specific state. The object may recursively require other objects as parameters and the typing must match. Tonella [2004] solved this problem by defining a grammar for the chromosome and defining the mutation and crossover operations based on it. Another problem when generating test cases for object-oriented program is the lack of software specification to check if a test has passed. Wappler and Wegener [2006a] used software exceptions as an indication of a fault and Alander et al. [1998] used the time needed for execution.

In particular several issues in the current state-of-art of test data generation for object-oriented software can be pointed out, namely:

- little work has been done using optimization algorithms.
- empirical tests have always been performed on a very small clusters of classes, this causes to reduce the reliability of the results.
- no common benchmark cluster exists which can be used to test and compare the different techniques.
- there are no comparisons between different optimization algorithms on the testing of the same classes.

1.7 Research Agenda

Much research has been done in the object oriented area which has been involved with the development and evaluation of quality software products. The structural attribute measures which intended to quantify important characteristics of object-oriented software, such as size, polymorphism, inheritance, coupling, and cohesion are needed to relate with the external quality indicators such as fault proneness, change impact, reusability, development effort and maintenance effort. Much study has been done in literature to study the relationships between coupling and cohesion with external quality factors of object-oriented software. Clear empirical relationships have been identified between class level coupling and fault proneness of the class. The purpose of this research is to investigate a special type of coupling namely indirect coupling and its relationship to maintenance effort and to explore the nature of this relationship so that better quality software can be produced having good maintainability. The quality of developed software must be tested by executing it with good test cases, which increases the confidence of a developer in developed software. In this research it is also investigated that how genetic algorithms can be applied to generate better test cases for object oriented software. This research will try to find answers to following research questions:

How is Software Quality related to coupling between classes?

Maintainability of software is a key quality concept while coupling is a way to determine how independent a class is from the others in the software. If a class has strong coupling with other classes then it increases the complexity of the software and it becomes less maintainable, which

ultimately affects its overall quality. However, study of various forms of coupling and their impact on quality metrics is still a topic of research and the presented research explores this relation to a greater detail.

What is indirect class coupling and how can it be measured?

A form of coupling that has so far received little attention is indirect coupling that is, coupling between entities that are not directly related. Indirect coupling through a class makes an indirect coupling path, the length and coupling through this path decides how strongly a class is coupled with other classes in the software. In the presented thesis we propose a metric which may be used to measure this form of coupling.

What are the possible relationships which may exist between indirect path coupling and class maintenance effort?

The indirect path coupling is based on the idea that the longer the path connecting two modules, the more hidden the dependencies are. Consequently it becomes more difficult to detect such indirect coupling. Furthermore, such coupling is expected to increase the overall effort when maintenance activities are performed. Through the work in this research we establish this relationship using the proposed metrics. We attempt to find mutual relationships between this maintenance effort and indirect coupling in this research.

How can the quality of class level test data be improved?

Object-oriented software development raises important and challenging class level testing issues that cannot be solved directly by existing testing techniques [Beizer 1990; Roper 1994; Beizer 1995] for conventional programming languages. The enhancement of the existing techniques is necessary, and new testing processes specific to object-oriented programs need to be established [Harrold and McGregor 1992; Smith and Robson 1992; Hoffman and Strooper 1993; Binder 1996]. Various test data generation techniques are described in literature [Baresel et al. 2002; Harman and McMinn 2007; Ferrer et al. 2012]. This research explores evolutionary testing techniques to be applied for test data generation.

How can fitness evaluation technique be improved in evolutionary testing?

A fitness function guides search into promising, unevaluated areas of the search space. In evolutionary algorithms formulation of test goal is achieved through designing a good fitness function. This research focuses on a more effective fitness evaluation technique which can be used to generate better test cases.

Can we devise a technique to generate test sequences that increases feasible coverage using a genetic algorithm?

In search based test data generation, test data is generated to meet the requirements of a particular test adequacy criterion like coverage. The generation of maximum feasible test cases to meet such criteria is always a challenge. In this thesis we propose a technique to increase the feasible test cases to maximize the selected coverage criterion.

1.8 Organization of the Thesis

A brief introduction to software quality, metrics for object oriented software, software testing, test automation and the research questions investigated in this thesis are discussed in this introductory chapter. The metrics for object oriented software; features of software test data generation and importance of measurement and testing of software have also been discussed in this chapter. State-of-the-art of the metrics and software test data generation approaches are also discussed briefly.

In chapter 2, detailed literature survey has been presented on object oriented software metrics, measurement and testing. Various metrics models, various techniques in object-oriented approach to the design of testing strategies are discussed. Several testing issues are unique to O-O software, research which empirically explores the relationships between existing object-oriented coupling, cohesion, and inheritance measures and the probability of fault detection in system classes during testing are explored. The difficulties in automated test data generation, various techniques are surveyed.

Chapter 3 explains the about code and design metrics, reusability, maintainability and their relationship with coupling and cohesion. Various types of couplings are explained. The concept of indirect coupling is elaborated and the metrics used to measure indirect coupling are decribed. Relation of indirect coupling with software maintainability has also been discussed. This chapter also

explains software testing and various levels of software testing. We discuss why and how the evolutionary techniques could be applied at class level.

Chapter 4 describes the methodology chosen for this thesis.It first explores the research questions and theninvestigates the steps taken to answer the research questions by describingtheuseofthemethods which are the most appropriate, given the aims and nature of the research.

Chapter 5 discusses the proposed indirect coupling metric. Simple Transitive form of indirect coupling has been considered and experiments are conducted to correlate with maintenance effort. Different versions of open source software are considered and the proposed metric is measured using various tools and it is correlated with maintenance effort which has been derived from the analysis of source code of different versions of the softwares taken in case study.

Chapter 6 explains the automated techniques to generate test cases at class level. A technique of test case generation for object oriented software class has been proposed by applying finite state machine specification and genetic algorithms. It also shows that genetic algorithms are useful in reducing the number of infeasible test cases by generating test cases for object oriented software. Furthermore, it has been established that the proposed technique is good for structural testing for generating more suitable test cases.

Chapter 7 concludes the thesis by summarizing the achievements of the work and provides limitations, critical analysis of the work and suggests further investigations of the presented work.

Following the chapter 7, the List of References, Appendix A, Appendix B, List of publications by author and biosketches of the author & his supervisor are appended.

CHAPTER 2 *Software Quality Metrics and Test Data generation – A Literature Review*

The measurements represented by a quality metrics can be obtained during all phases of the software development to provide an indication of progress towards the desired product quality. Various metrics have been proposed for measuring properties of object-oriented software such as size, complexity, cohesion and coupling. These object-oriented metrics can be used as significant predictors for the maintainability of software. Also to gain confidence in quality of the software, various methods exist, for example static analysis, code reviews, formal specifications, refinement and proof. However, testing remains the primary method [DeMillo and Offutt 1991], particularly in industry.

This chapter provides a survey of object oriented metric models, software quality measurement approaches and techniques used to guide the selection of test data. Suitable application of these measurements and testing techniques are critical components of any high quality software.

2.1 Estimation of Software Quality

Since the last few decades in software engineering research area the estimation of software quality has become one of the most interesting research area. Software quality estimation is used to identify the errors that might have been introduced during software development cycle because the cost can be reduced significantly if the errors could be identified at earlier stages and therefore can help to enhance software quality.

Since quality of software directly affects the application and maintenance of software therefore it is important to consider methods scientifically evaluate the software quality [Yang and Zhang 2009]. The evaluation of software quality involves various activities to be carried out during software development process. These are continuous measurement of quality throughout development process, determining current status of software, prediction of follow up development trend of software quality and provide effective means for developer, customer and evaluator. Generally some set of evaluation activities are defined in software quality specification of project plan as well as related specifications

for software quality. The developer may apply quality evaluation on the finished product as well as on semi finished products at each phase of development by identifying the difference between current quality level and the required quality level of product, by taking timely corrective actions. It ensures the quality to be incorporated at each level of development and product can meet the final level of quality requirements.

Software quality is measured with the help of several quality factors. Software coupling which designates the way in which various software components are connected, may be used to estimate a number of quality factors, including maintainability, complexity and reliability [Offutt et al. 2008]. Since long time software coupling has been used to evaluate software. The coupling information can be obtained from design document before software could be implemented or even from the source after it has been implemented. Both ways have their merits and demerits. If coupling information id obtained before implementation then it the information and measurements could be used in project planning, implementation and test preparation. On the other hand coupling information obtained after implementation can help to incorporate decisions and reflects changes made during the implementation.

2.2 Software Quality Metric Models and Maintainability

Large amount of research has been done during last four decades for determining measurable properties and defining mathematical relationships between product qualities and process quantities. A quantitative study has much importance to be able to assess, predict and control software process characteristics. The Object Oriented (OO) approach is based upon modeling the real world in terms of its objects. This approach differs from traditional approach which emphasizes a function oriented view where data and procedures are separated. Therefore the views in these two notions are fundamentally different and that’s why the metrics developed for traditional methods do not work with classes, encapsulation, inheritance or dynamic binding.

The study reported in [Boehm et al. 1976] establishes a conceptual framework and some key initial results in the analysis of the characteristics of software quality. It shows that explicit attention to characteristics of software quality can lead to significant savings in software life-cycle costs. Adefinitive hierarchy of well-defined, well-differentiated characteristics of software quality is

developed. Various software quality-evaluation metrics have been defined, classified, and evaluated according to their potential benefits, quantifiability, and ease of automation.

Later on, working on a framework for software quality measurement, Cavano and McCall [1978] describe a quality framework which can potentially provide significant benefits to SQA programs. It enforces a life cycle management viewpoint on quality assurance activities and provides early indications of quality problems. Since formal relationships between the metrics and their related quality factors have not been validated so they suggests that there are indications based on a limited sample that such relationships can be established. Since the software quality cannot be measured directly, software metrics are used to measure the parameters which establish software quality. There are basically two general types of criticism applicable to current software metrics. The first category is associated with conventional software metrics as they are applied to traditional, non object-oriented software design and development. Kearney [Kearney et al. 1986] criticized the existing software complexity metrics in their research for being without solid theoretical bases and lacking appropriate properties. Vessey and Weber [1984] also commented on the general lack of indepth theoretical work in the structured programming literature. According to Prather [1984] and Weyuker [1988], the traditional software complexity metrics do not possess appropriate mathematical properties, and therefore they consequently fail to display the normal predictable behavior. There is also a second category of criticism which is about Object Oriented (OO) design and development. Basically the OO approach involves modeling the objects of the real world, while more traditional approaches emphasize a function-oriented view where data and procedure is treated separately. Chidamber and Kemerer [1994] argue that, because these two approaches have fundamentally different inherent notions, the software metrics which are developed with traditional methods keeping in mind are not appropriate for the notions such as classes, inheritance, encapsulation, and message passing. Therefore it can be said that current software metrics are subject to some general criticism and are easily seen as not supporting key OO concepts, it seems appropriate to have new validated metrics which are especially designed to measure the unique aspects of the OO design [Abreu and Carapuca 1994; Abreu and Fernando 1995; Basili et al. 1996; Lorenz and Kidd 1994].

Study by Kafura and Reddy [1987] relates seven different software complexity metrics to the experience of maintenance activities performed on a medium size software system. Three different

versions of the system that evolved over a period of three years were analyzed in this study. Several theoretical discussions [Booch 1991; Kim and Lerch 1991] have speculated that OO approaches may even induce different problem-solving behavior and cognitive processing in the design process. Because of the fundamentally different notions inherent in these two views, Wilde and Huitt [1992] find that software metrics which are developed with traditional methods in mind do not readily find place with OO notions such as classes, inheritance, encapsulation and message passing.

At the same time Fenton's work [Fenton 1991] categorizes software measures along two orthogonal axes. The first is the process/product axis: a metric may measure an attribute of software product, (e.g., quality of code), or an attribute of software development process (e. g., cost of design review meetings). Another, orthogonal axis is the internal/external axis. A metric may measure an internal attribute (e. g., the number of loops in a module), or an external attribute (e. g., maintainability of a module). Because of the shortcomings of existing metrics, new metrics which are especially designed for OO software are suggested by various authors. Tegarden [Tegarden et al. 1992] and Bilow [Bilow 1992] have suggested for a theoretical research in the design of OO metrics. Some initial proposals for such kind of metrics are referred by Morris [1988], although they are not tested. A more formal attempt is performed by Lieberherr and his colleagues [Lieberherr et al. 1988] who present at defining the rules of correct object oriented programming style, building on concepts of coupling and cohesion that are used in traditional programming. Similarily Coplien [1993] suggests a number of rules of thumb for OO programming in C++. Three metrics for OO graphical information systems are suggested by Moreau and Dominick [1989], but they do not provide formal, testable definitions. The need for new measures is also suggested by Pfleeger [Pfleeger et al. 1990], and they use simple counts of objects and methods to develop and test a cost estimation model for OO development. Metrics for measurement of inheritance in C++ environments is prescribed by Lake and Cook [1994], and they have gathered data from an experimental system using an automated tool. The various other authors [Chidamber and Kemerer 1991; Sheetz et al. 1992; Whitmire 1992] propose metrics which are based primarily upon pragmatic insights and recommendations from [Lorenz and Kidd 1994], but they do not offer any empirical data. Also Rajaraman and Lyu [1992] and Li and Henry [1993] test the metrics proposed in [Chidamber and Kemerer 1991] and measured them for applications developed by university students. Despite of having the active interest in this area, no empirical metrics data from commercial object oriented applications have been published in

the archival literature. The earliest measures which were based on code analysis; the most fundamental was count of the number of Lines of Code (or LoC). Although various researchers criticize it for various reasons it remained as a measure of complexity because of its simplicity.

The CK metrics [Chidamber and Kemerer 1994] are empirically investigated by Basili et al. [1996] and results of this study are presented to assess these metrics as predictors of fault-prone classes and, therefore, determine whether they can be used as early quality indicators. Several of Chidamber and Kemerer's OO metrics [Chidamber and Kemerer 1994] appear to be useful to predict class fault-proneness during the early phases of the life-cycle.

The ways to test for the essential OO language features of inheritance and polymorphism was looked for by the researchers, finally. Several different ideas have been put forward, each with advantages and disadvantages. [Murphy et al. 1994] established that using object-oriented technology, constructing a highly available, stable and robust system requires new techniques and tools to test the software, particularly at the class level. Tool support for object-oriented testing is also discussed including specification editing, test cases generation, and test cases execution and validation.

Gulezian [1991] addresses the development of a unified view and structure for a proactive approach toward software quality improvement during development. He has shown that by introducing a reformulated concept of process maturity which is strictly measurement-oriented, it is possible to provide a proper reference frame and a clearer guide toward requirements relating to implementation.

For this purpose a model for software product quality is defined in [Dromey 1995]. A set of quality-carrying properties have been associated with each of the structural forms which are used to define statements and statement components of a programming language. These properties having quality information are in turn linked to the high-level quality attributes as specified by the International Standard for Software Product Evaluation ISO 9126. The model given by Dromey [1995] also supports the incorporation of quality into software, systematically classifying quality defects, definition of language-specific coding standards and the development of automated code auditors for detecting defects in software.

The results of an investigation into a set of metrics for object-oriented design, called the MOOD metrics is described in [Harrison et al. 1998]. A measurement theory viewpoint has been considered for the merits of each of the six MOOD metrics, which takes into account the recognized object-oriented features which Harrison et al. [1998] were intended to measure: encapsulation, inheritance, coupling, and polymorphism. The relationships between existing object-oriented coupling, cohesion, and inheritance measures and the probability of fault detection have been explored empirically by Briand et al. [1998] in system classes during testing. Results show that many of the measures capture similar dimensions in the data set, thus reflecting the fact that many of them are based on similar principles and hypotheses. For fault-proneness, the method invocations frequency and the depth of inheritance hierarchies are also the main driving factors oher than the size of classes.

Many organizations want to predict the number of defects or faults in software systems, before they are deployed, to gauge the likely delivered quality and maintenance effort. For this purpose statistical models are developed. Authors [Fenton and Neil 1999] argue that such models are weak because of their inability to cope with the unknown relationship between defects and failures. They recommend holistic models for software defect prediction, using Bayesian Belief Networks, as alternative approaches to the single-issue models used at present. [Fenton and Neil 1999] also argue for research into a theory of "software decomposition" in order to test hypotheses about defect introduction and help construct a better science of software engineering.

The study by Tang [Tang et al. 1999] investigates the correlation between object-oriented design metrics and the likelihood of the occurrence of object oriented faults. The CK metrics suite [Chidamber and Kemerer 1994] is validated using these faults. They also proposed a set of new metrics that can serve as an indicator of how strongly object-oriented a program is, so that the decision to adopt object oriented testing techniques can be made, to achieve more reliable testing and also minimize redundant testing efforts. A criterion for analyzing and testing the polymorphic relationships that occur in object-oriented software is described by Alexander and Offutt [1999]. The techniques adapt traditional data flow coverage criteria to consider definitions and uses among state variables of classes, particularly in the presence of inheritance and polymorphic overriding of state variables and methods. By applying these criteria an increased ability to find faults can be resulted and they create higher quality software.

In order to provide a quantitative approach to relate measurable object-oriented characteristics to the higher-level desirable software quality attributes, Bansiya [2002] extended the Dromey's generic quality model [Dromey 1995] to propose a hierarchical model for an object-oriented design quality assessment approach, called QMOOD. Godfrey and German [2008] discuss the concept of software evolution from several perspectives. They have examined how it relates to and differs from software maintenance. They discuss insights about software evolution arising from Lehman's laws of software evolution and the staged lifecycle model of Rajlich and Bennett [2000].

There are several studies which have identified clear empirical relationships between class-level coupling and the fault-proneness of the classes. Static code analysis is a common way to quantify the coupling. However, the resulting static coupling measures only capture certain underlying dimensions of coupling. There exist some other dependencies regarding the dynamic behavior of software they can only be inferred from run-time information. Arisholm [2002] describes how several dimensions of dynamic coupling can be calculated by tracing the flow of messages between objects at run-time. Arisholm [2002] also describes a simple algorithm for collecting the measures.

Maintainability of a program can be enhanced by using program refactoring technique. A quantitative evaluation method has been proposed by Kataoka et al. [2002] to measure the maintainability enhancement effect of program refactoring. They focused on the coupling metrics to evaluate the refactoring effect. The degree of maintainability enhancement could be evaluated by comparing the coupling before and after the refactoring. It was shown that their method was effective to quantify the refactoring effect and helped to choose appropriate refactoring. Table 2.1 presents a list of software maintainability predictors reported in various studies, which are measured at source code level or project's post-implementation stage [Riaz et al. 2009]. Similarly the Table 2.2 summarizes various maintainability metrics defined in literature.

Marinescu [2005] defines a detection strategy mechanism so that deviations from good-design principles and heuristics can be quantified in form of metrics-based rules. Classes or methods affected by a particular design flaw (e.g. God Class) can directly be localized by using detection strategies by an engineer, rather than having to infer the real design problem from a large set of

abnormal metric values. Marinescu [2005] proposes a quality model, called Factor-Strategy which relates explicitly the quality of a design to its conformance with a set of essential principles, rules and heuristics, which are quantified using detection strategies. Ostrand et al. [2005] used historical data from two large software systems with up to 17 releases to predict the files with the highest defect density in the following release. A negative binomial regression model has been developed and used to predict the expected number of faults in each file of the next release of a system. For each release, the 20% of the files with the highest predicted number of defects contained between 71% and 92% of the defects being detected.

Table 2.1 software maintainability predictors gathered at source code level [Ostrand et al. 2005]

S.No.	Metrics	S.No.	Metrics
1	Halstead's Effort 'E'	23	Mean tokens per method
2	Avg. effort per module 'ave-E'	24	Mean no. of decisions per method
3	Halstead's volume 'V'	25	Lack of Cohesion of Methods 'LCOM'
4	Lines of code 'LOC', 'SIZE1'	26	Halstead's difficulty 'D'
5	Halstead's predicted length 'N^'	27	Data Abstraction Coupling 'DAC'
6	Avg. volume per module 'ave-V'	28	Halstead's program vocabulary 'n'
7	Avg. no. of comment lines per module 'ave-CMT'	29	Halstead's unique operands count 'n2'
8	Avg. LOC per module 'ave-LOC'	30	Halstead's unique operands count 'n1'
9	Avg. extended cyclomatic complexity per module 'ave-V(g)'	31	Maximum number of operations
10	Avg. predicted length per module 'ave-N^'	32	No. of overridden methods
11	Avg. length per module 'ave-N'	33	Percentage of private member
12	extended cyclomatic complexity 'V(G)'	34	Depth of Inheritance Tree 'DIT'
13	Total purity ratio 'PR'	35	No. of Children 'NOC'
14	Avg. purity ratio per module 'ave-PR'	36	Response For a Class 'RFC'
15	Sum of the avg. variable spans 'SPN'	37	Weighted Method per Class-'WMC'
16	Total no. of executable semicolons 'NES'	38	Message Passing Couple 'MPC'
17	Number of comment lines 'CMT	39	No. Of Methods 'NOM'
18	average variable span per module 'ave-SPN'	40	Number of properties 'SIZE2'
19	Avg. no. of executable semicolons 'ave-NES'	41	Avg. complexity per method 'OSAVG'
20	cyclomatic complexity	42	Avg. no. of methods per class 'CSO'
21	Median lines of code per object method	43	Avg. attributes per class 'CSA'
22	Type of object	44	Avg. no. of children per class 'SNOC'

Software coupling can be used to estimate a number of quality factors, including maintainability, complexity, and reliability. [Offutt et al. 2008] discusses software couplings based on object-oriented relationships between classes, which specifically focuses on types of couplings that are not available until after the implementation is completed, and a static analysis tool is presented which measures couplings among classes in Java packages. Design documents can be used to derive coupling information which allows the information to be available earlier (pre-implementation) and therefore it is more useful for predictive purposes, whereas derivation of coupling information from source code allows the information to be more precise and it reflects decisions which were made during implementation and are not specified in the design documents.

Table 2.2 Summary of Maintainability Metrics [Ostrand et al. 2005]

Authors	Measures/Metrics for Maintainability
[Oman and Hagemeister 1994]	Subjective assessment (ordinal scale metric) by using the U.S Air Force Operational Test and Evaluation Center's AFOTEC software maintainability evaluation instrument, which provides a rating as well as categorizes maintainability as low, medium or high.
[Schneberger 1997] [Genero et al. 2001]	Expert opinion using an ordinal scale
[Misra 2005] [Zhou and Xu 2008]	Maintainability Index (MI)
[Lim et al. 2005]	1. Total effort in person-minutes to comprehend, modify, and test the artifacts related to the system 2. Volume of changes made to software artifacts. Change volume was measured as: a. The number of pages changed in a document b. The number of modified executable LOC c. The number of test cases constructed d. The LOC written to construct test scripts e. The number of files needed to be compiled f. The number of files needed to be deployed
[Koten and Gray 2006] [Zhou and Leung 2007]	CHANGE metric: count of LOC changed during a 3-year maintenance period.
[Shibata et al. 2007]	G(u) = 1–exp(- v^u), according to $M_t/M/\infty$ framework

Offutt et al. [2008] calculated the Spearman rank correlation between the various selected quantities which is shown in Table 2.3. It was found that there was statistically significant correlation at the 95% level of confidence between the number of lines of code (LOC) and the number of instances of each type of coupling (parameter coupling per class (PCC), Global coupling (GCC), and inheritance

coupling per class (ICC)). Therefore it concludes that, the more code, the more is the coupling. But from the observation it can be noted that this does not mean one can use LOC to infer the amount of coupling.

Table 2.3 Correlations in terms of Spearman rank correlation *R* values [Offutt et al. 2008]

Spearman rank correlation	NOC	LOC	PCC	GCC	ICC	TCC
NOC	1	0.597	0.542	0.603	0.792	0.542
LOC		1	0.982	0.875	0.767	0.982
PCC			1	0.834	0.767	1
GCC				1	0.558	0.834
ICC					1	0.767
TCC						1

The fact which was deduced from above table 2.3 indicates that NOC was not correlated with parameter couplings. It is interesting finding. This lack of correlation leads the authors to deduce that, for the sample of 11 open-source Java projects they had examined, parameter coupling is primarily used within classes, whereas global and inheritance coupling has been primarily used among classes [Offutt et al. 2008].

2.3 Software Test Data Generation Techniques

Software test-data generation is the process of identifying a set of data, which satisfies a given testing criterion. For solving this difficult problem a lot of research work has been done in the past. The most commonly encountered are random test-data generation, symbolic test-data generation, dynamic test-data generation, and recently, test-data generation based on genetic algorithms. Sultan et al. [2010] gives a survey of the majority of software test-data generation techniques based on genetic algorithms. They also compare and classify the surveyed techniques according to the genetic algorithms features and parameters. Also, their research shows and classifies the limitations of these techniques.

Several techniques proposed in the object-oriented approach on the design of testing strategies are reviewed in [Gu et al. 1994]. In particular, they focus on the test case selection problem in class testing and investigate the impact of object-oriented approach on the design of testing strategies. Due to the nature of software faults, in any software testing, a software fault is exposed only when the statement where the fault resides is executed under certain operational environment. For a class, its

state space is a major component in its operational environment. Detailed state of the art in object oriented software testing is described in [Binder 1996]. Several testing issues are unique to OO software. A number of researchers have asserted that some traditional testing techniques are not effective for object-oriented software [Berard 1994; Firesmith 1993; Gu et al. 1994] and that traditional software testing methods test the wrong things. Researchers have been developing new methods and techniques to test OO software for a number of years. Early work focused on testing of data abstraction and state behavior [Cheatham and Mellinger 1990; Firesmith 1993]. Subsequent work looked into testing of classes and issues such as what kind and how many objects should be instantiated and the order in which classes should be tested [Chen et al. 1998; Harrold and McGregor 1992]. More recently, researchers have looked at integration issues of OO software and testing of complete classes [Gallagher and Offutt 2004].

A technique which generates test cases for class-level object-oriented software testing and integrates the testing techniques based on algebraic specifications, model-based specifications, and finite state machines is given by Tse and Xu [1996]. The formal object-oriented specifications are used to guide the test case construction process. In this approach, testers first analyze the formal specification of a class to partition the state space of the class. It requires to identify a test model that is based on finite-state machines, then the class specification needs to be analyzed and the test model is used to select a set of test data for each method of the class, and finally the test cases of the class are prepared from the test model by following various well-developed testing criteria.

Chen et al. [2000] proposed an integrated approach for selecting fundamental pairs of equivalent ground terms as class-level test cases for object-oriented programs and applying observable context technique to determine whether the objects resulting from the execution of a test case are observationally equivalent. It first utilizes the fundamental pair approach of Chen et al. [1998], combined with regularity and uniformity hypotheses, to generate a finite number of test cases. To determine observational equivalence of objects resulting from the set of fundamental pairs, a series of methods called the relevant observable context is constructed from the implementation of a given algebraic specification.

[Briand et al. 2002] presents an improved strategy to devise optimal integration test orders in object-oriented systems. Minimizing the complexity of stubbing during integration testing is important as this has been shown to be a major source of expenditure. Their strategy to do so is based on the combined use of inter-class coupling measurement and genetic algorithms. The former is used to assess the complexity of stubs and the latter is used to minimize complex cost functions based on coupling measurement.

An automated and simplified genetic programming (GP) based decision tree modeling technique is presented by Liu [Liu and Khoshgoftaar 2003] for calibrating the software quality classification models. This technique is based on multi-objective optimization using strongly typed GP. To optimize the classification accuracy, two fitness functions are used and tree size of the classification models is calibrated for a real-world high-assurance software system. They have shown that the GP-based decision tree technique yielded better classification models. Khoshgoftaar [Khoshgoftaar et al. 2000] in their paper present an empirical case study to show how principal components analysis can improve a classification-tree model.

Some techniques use optimization based techniques (e.g. genetic algorithms) for automated test case generation. In [Tonella 2004] a genetic algorithm has been used to automatically produce test cases for the unit testing of classes in a generic usage scenario. In this approach the test cases are represented by chromosomes, which include information about the objects which are to be created, the methods to be invoked and values which are used as inputs. The proposed algorithm in [Tonella 2004] mutates the represented chromosomes with the aim of maximizing a given coverage measure. Tonella [2004] used genetic algorithms for generating unit tests of Java programs. Solutions are modeled as sequences of function calls with their inputs and caller (an object instance or a class if the method is static). The literature proposes special crossover and mutation operators to enforce the feasibility of the generated solutions. Similar work with genetic algorithms has been done by Wappler and Lammermann [2005], but they used standard evolutionary operators. Infeasible individuals may be resulted in this approach, which are penalized by the fitness function. Mcminn [2004] has shown that Evolutionary Testing can be successful in automatically generating relevant unit test cases for procedural software. Applying these techniques can increase efficiency and quality of the usually costly test data generation process [Sthamer et al. 2002]. Liu et al. [2005] use a hybrid

approach, in which ant colony optimization is exploited to optimize the sequence of function calls. Multi-agent Genetic Algorithm is then used to optimize the input parameters of those function calls.

Object-oriented unit tests consist of sequences of method invocations. The response of an invocation depends on the method's arguments and the state of the receiver at the beginning of the invocation. Also there are two tasks involved in generating unit tests: the first is generating method sequences that build relevant receiver object states and the second is generating relevant method arguments. [Xie et al. 2005] proposes a framework that achieves both test generation tasks using symbolic execution of method sequences with symbolic arguments. Their paper defines symbolic states of object-oriented programs and comparisons of states. Given a set of methods from the class under test and a bound on the length of sequences, the framework systematically explores the object-state space of the class and prunes this exploration based on the state comparisons. Software metrics can provide us with information regarding the quality of software. The ripple effect metric shows what impact changes to software will have on the rest of the system. It can be used during software maintenance to keep the system at a high level of quality. The research in [Bilal and Black 2006] focuses on implementing ripple effect measurement for object oriented software.

In literature the researchers have successfully applied evolutionary algorithms to software testing. Method call sequences that realize interesting test scenarios are required to be generated for testing object oriented programs. Any arbitrary method call sequence is not necessarily feasible due to call dependences which exist among the methods that potentially appear in a method call sequence. Wappler and Wegener [2006b] described an approach to automatically generating test cases for structure-oriented unit testing of object oriented software. They have used strongly-typed genetic programming for the generation of method call sequences. The generation of method call sequence forms the basis of the test cases.

The empirical evidence for the effectiveness of evolutionary testing consists largely of small scale laboratory studies. Harman and McMinn [2007] present a first theoretical analysis of the scenarios in which evolutionary algorithms is suitable for structural test case generation. In their work, this theoretical analysis is supported by an empirical study which considers real world programs and the search spaces which are several orders of magnitude larger than those considered earlier.

There are two main challenges in object-oriented programs while achieving high structural coverage such as branch coverage. First, some branches involve complex program logics and for testing such branches the testing logic require deep knowledge of the program structure and semantics to generate tests to cover them. Second, covering some branches requires special method sequences to lead the receiver object or non-primitive arguments to specific desirable states. Therefore, to overcome problems of earlier approaches Inkumsah and Xie [2007] propose a framework called Evacon that integrates evolutionary testing (used to search for desirable method sequences) and concolic testing (used to generate desirable method arguments). Their experimental results show that the tests generated using their framework can achieve higher branch coverage than evolutionary testing or concolic testing alone.

While performing evolutionary testing the nested predicates can cause problems, because information needed for guiding the search only becomes available as each nested condition is satisfied. This means that the search process can overfit to early information, making it harder, and sometimes nearly impossible, to satisfy constraints and they become apparent later in the search. In [McMinn et al. 2009] a testability transformation is presented that allows the evaluation of all nested conditionals at once.

Lochmann [2010] presented an approach that relies on a quality model, that defines software quality and that serves as a structured knowledge-base. This quality model is integrated with a use-case based approach for eliciting and analyzing quality requirements. This way an effective communication with stakeholders as well as the quantification of quality requirements can be assured.

2.4 Research Gaps

In the view of the literature review we identify the following research gaps:

1. There is a lack of quantitative measures of relationship between the software maintainability and the internal characteristics of the software.
2. Most of existing metrics are not intuitive. It requires education on the users' side to have numerical thinking about the quality of software and how to apply them.

3. Some internal attributes of the software are still to be explored to understand their relationship with quality.
4. Existing definitions of coupling do not capture the full essence of the original notion posed by Yourdon and Constantine [1979].
5. Existing approaches to test at unit level of object oriented software are not much effective in terms of achievable structural coverage.
6. Design defects within a class are discussed more in literature than design defects of a class cluster and at behavioral level.
7. There is a lack of the availability of standard testing process measurement method for evaluating the effectiveness of a test process.
8. There is inefficiency in test generation process due to infeasible test sequences.

2.5 Objectives of the Research

To reduce some of these gaps, the present research is aimed at:

- Defining a simple object-oriented metric to quantify the maintainability of object oriented software classes. The metric should be able to relate software maintenance with internal product characteristics.
- Developing an approach to automate the software testing process to enable the generation of test sequences that can create arbitrary objects that serve as arguments for succeeding methods. The approach should allow the application of automatic test generation to classes having primitive as well as object type arguments.
- Design objective function which can provide sufficient guidance even in presence of runtime exceptions, which prematurely terminate the evaluation of a test case.

To target these aims, there has been an intensive literature survey on software quality metrics and its measurements from various journals, web sites, books, conference publications etc. A greater emphasis is given on the software test generation techniques at unit level testing of object oriented software.

In this chapter we have discussed how from their first appearance, automatically computable metrics have become an important tool for assessing attributes of software and software-related activities.

Metrics are characterized in different categories like process metrics for measuring characteristics of software development processes, product metrics for assessing software products, and resource metrics for measuring characteristics of software-related resources. Maintainability is an important aspect of software quality and is highly related to complexity of large software systems. We have discussed the work done which relates maintainability with coupling between its modules and size. We also discussed work done to automate the test data generation using evolutionary techniques.

The next chapter discusses the important theory of software maintenance, maintainability metrics, coupling, and evolutionary techniques to generate automated test data in software testing.

CHAPTER 3 *Software Maintenance and Test Case Generation*

There has been an increased awareness in recent years of the critical problems that have been encountered in the development of large scale software systems. These problems not only include the cost and schedule overruns typical of development efforts, and the poor performance of the systems once they are delivered, but also include the high cost of maintaining the systems, the lack of portability, and the high sensitivity to changes in requirements. The potential of the software metric concepts can be realized by their inclusion in SQA. Their impact on quality assurance is to provide a more disciplined, engineering approach to quality assurance and to provide a mechanism for taking a life cycle viewpoint of software quality. The benefits derived from their application are realized in life cycle cost reduction.

The actual measurement of software quality is accomplished by applying software metrics (or measurements) to the design and source code produced during a software development. These measurements are part of the established model of software quality and through that model it can be related to various user oriented aspects of software quality. The measurement concepts complement Quality Assurance and testing practices. They are not a replacement for any current techniques utilized in normal quality assurance programs. For example, a major objective of quality assurance is to assure conformance with user requirements. The software metrics add formality and quantification to design and code developed during software life cycle. When one talks about the Object-Oriented (OO) paradigm which has grew out of a need to meet the challenges of past practices using standard structured programming, becoming clear about these challenges, one can better understand the advantages of OO programming, as well as gain a better understanding of this mechanism. Testing of OO software is different from testing software created using procedural languages. Several new challenges are posed and new techniques have been developed.

3.1 Quality Factors for Object-Oriented Paradigm

OO design and development is a popular concept in today's environment for software development. They are often referred to as best solution to software problems. Although in reality it is not miraculous, OO technology has proven its value for systems that undergo maintenance and changes often after deployment.Since their introductionin 1970, software metrics which could be computed automatically using tools for assessment of software attributes have become important. Fenton [1991] has categorized these metrics as follows:

- *process metrics* for measuring characteristics of software development processes.
- *product metrics* for assessing software products such as components, procedures and programs.
- *resource metrics* for measuring characteristics of software-related resources such as hardware and personnel.

Fenton also differentiated between internal and external metrics. Internal software metrics are those which are used to measure software attributes which can be measured directly by examining the software code on its own irrespectively of its behavior. External product metrics are those used to measure attributes of the software that can be measured only with respect to how the software relates to its environment. Using this terminology, this work is focused on internal software metrics for OO software. Rocacher [1988] suggested that traditional metrics cannot be considered sufficient for assessing the attributes of OO software, because they are not designed to deal with the OO concepts like encapsulation, inheritance, polymorphism etc. One of the most important aspects of OO metrics is its ability to focus on the combination of function and data in terms of integrated objects, while, traditional metrics measure the design structures and data structures independently.

Whenever we consider the quality factors exercised during implementation and maintenance phase, this requires focusing on quality aspects like maintainability and reusability. In literature this is generally expressed in terms of a hierarchy of factors and criteria [Frappier et al. 1994]. The factors which are assigned in higher level of this hierarchy generally represent the management's view point while the lover level factors are related to the measurement issues of the software code. The quality factors related to software code have an important place in defining the overall software quality. Measuring such metrics can represent the software quality progress towards the desired level.

Reusability and maintainability such two factors which are a good indication of software quality used during software development and applied on code.

Chowdhury and Zulkernine [2010] related the above quality factors reusability and maintainability with the code related metrics which are Complexity, Coupling, and Cohesion and investigated whether these metrics can be utilized as early indicators of software vulnerabilities. It was found by them that these metrics are correlated to vulnerabilities at a statistically significant level. Therefore we can say that means these metrics are direct indicators of vulnerabilities and if there are vulnerabilities it will require more effort during maintenance. Hence it suggests that there must be some relationship between the above metrics and maintenance effort. These metrics can provide a measure of maintenance effort in terms of coupling and cohesion. This work focuses on indirect form of software coupling metrics and analyzed its relationship with the maintenance effort.

3.2 Code and Design Metrics

Chidamber and Kemerer are two of the leading researchers in software metrics and have introduced their work by providing a basic suite for colleting OO code and design metrics called as CK metric suite [Chidamber and Kemerer 1991]. Their research claims to help designers and managers in better decision making by using several of their metrics collectively. A significant amount of interest has been generated by CK metrics and these are currently the most well known suite of measurements for OO software [Fenton and Pfleeger 1998]. Measures for coupling and cohesion are included in CK metrics suite, the suite provides descriptive power for administrative concern. Mainly high level of coupling and low level of cohesion are associated with maintainability problems. For example, CBO is the number of other class with which a class is coupled. Class Coupling Metrics (CCM) measures the coupling between class and other class; Message Passing Coupling (MPC) measures the complexity of message passing between classes as well as objects. In object oriented programs the messages are passed among objects, but the types of messages passed are defined is class.

Abreu and Fernando [1995] defined Metrics for Object Oriented Design (MOOD) metrics. A basic structural mechanism of the OO paradigm such as encapsulation, inheritance, polymorphism, and message passing is referred by MOOD metrics. In this metric suite, each metric is expressed as a

measure where the numerator represents the actual use of one of those feature for a given design. MOOD metrics model uses two main features in every metrics which are methods and attributes.

The first quality model was proposed by McCall et al. [1977]. They presented a Software Quality Factor Framework and classified the quality attributes into three groups namely Product Revision, Product Operation and Product Transition. In this quality model McCall attempts to bridge the gap between users and developers by focusing on a number of software quality factors that reflect both the user's views and the developers' priorities. Figure 3.1 shows how he mapped quality factors Maintainability and Reusability to corresponding quality criteria.

3.2.1 Software Maintainability and Maintenance

Software maintenance forms an essential component of software development.
IEEE [IEEE 2006] defines software maintenance as:

> *The modification of a software product after delivery to correct faults, to improve performance or other attributes or to adapt the product to a modified environment.*

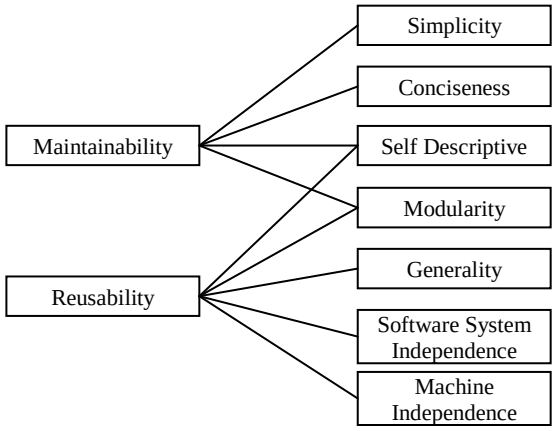


Figure 3.1 McCall's Model for Maintainability and Reusability

Software maintenance includes all the changes which are done after deployment. In this reference, maintainability refers to the easiness or toughness of the effort which is required to do the changes. Before any changes can be made to a software product, the It must be fully understood. It also requires testing the software thoroughly after the changes have been applied. For this reason, maintainability can be thought of as having three attributes: understandability, modifiability, and

testability. According to Harrison software complexity is the primary factor affecting these three attributes [Harrison et al. 1982], while modularity, information hiding, coupling, and cohesion are closely related to the complexity.

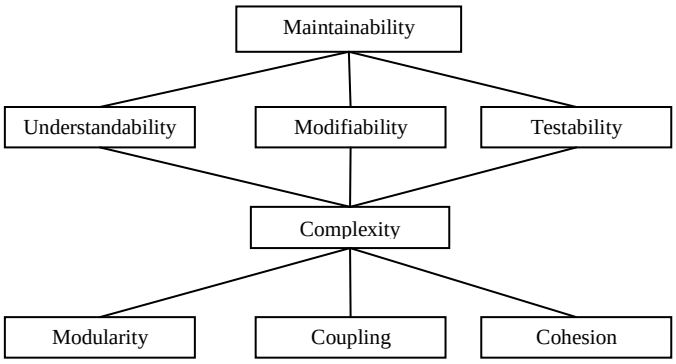


Figure 3.2 Maintainability factors

It's agreed that maintenance accounts for a large portion of software product's cost, if properly improved, it has a great potential to reduce the total software cost. If we want to improve maintainability we need to have a meaningful measure of maintainability [Harrison et al. 1982]. Although maintainability can only be accurately measured after the software product has been fully developed, previous work has shown that structural software attributes such as coupling and cohesion can influence maintainability of the final products [Briand et al. 2000; Card et al. 1986; Dagpinar and Jahnke 2003].

3.2.2 Reusability

Whenever it is required to write a reusable code, who is going to use it in what manner should be identified. If the code is having such a functionality which most of the users are expected to use then duplicating the code every time may result in such code which would be too expensive to produce and much difficult to use. This gives raise the need for code reuse but there are many difficulties associated with code reuse [Chang 2000]:

1. Code identification: Difficult to identify a piece of reusable code is termed as code identification difficulty. Many times, programmers reuse only a small fraction of their own or their colleagues' code.
2. Code validation and verification: The other difficulty is about correctness of the code. There is usually little assurance that the reused code is correct.

3. Code dependency: This task is associated to separate a desired piece of code from an entangled chunk of software having complex dependency.
4. Code modification: Even if the code is changed in new context, the reused code may implicitly conflict with the new context.
5. Execution environment: The assumptions used in reused code may be invalid in new environment. These assumptions may not be true in the new environment and therefore this may result in degraded performance.

This will require a careful planning and implementation which may avoid many of the above difficulties. This requires a reusable code to possess certain properties that can be measured using appropriate metrics. When implementation is completed just a static analysis of a source code may provide an instant feedback to the programmer, about the quality of the code in terms of reusability. This would encourage programmers to ensure that the completed code provides good reusability quality before it is discovered too late. The manager of a software project can also evaluate the quality using this measurement and can manage the project accordingly.

3.3 Metrics for Maintainability

Software maintainability is a difficult factor to quantify. Its measurement can be done indirectly by considering measures of design structure and software metrics. Logical complexity and program structure are claimed to have a strong correlation to the maintainability of the resultant software [Kafura and Reddy 1987; Rombach 1987]. Moreover, as Fenton says "*Good internal structure provides good external quality*". In order to define the maintainability of software product it must be determined that up to what point and in which cases can we rely on software metrics [Fenton 1991].

For measuring the maintainability of one specific product, the internal metrics are selected according to the nature of that specific product and the programming language used during its implementation. In order to estimate the level of the maintainability of a software product, the broadly used metrics like Halstead's software science metrics [Halstead 1975], cyclomatic complexity [McCabe 1982], Tsai's data structure complexity metrics [Tsai et al. 1986], lines of code, lines of comments, fan–in, fan–out, etc. can always be applied. To get a more reliable and acceptable measurement, the programs implemented in some specific type of programming language the software metrics which

can be applied only to that type should be used. For example, in the case of OO programming languages, the metrics that can also be used are: weighted methods per class, lack of cohesion of methods, coupling between objects, depth of inheritance tree, number of children, etc. [Hudli et al. 1994].

Achieving the desired level of quality is critical for all software development. Consequently, there is a large body of research in the area of software measurement in procedural and OO development, resulting in a large number of metrics having been proposed for measuring various aspects of the internal (structural) quality attributes of software such as coupling and cohesion [Baig 2004]. These structural attributes can be characterized by the following generic definitions: i) Coupling is a measure of the extent to which interdependencies exist between software modules; ii) Cohesion is the extent to which elements of a module contribute to one and only one task. In the following sub-sections, these metrics are described in some detail.

3.3.1 Coupling Metrics

Coupling is described as a "*measure of the strength of interconnection between modules*" [Yourdon and Constantine 1979]. The terms strength and interconnection have various interpretations, according to the various coupling metrics introduced in the literature. It is widely accepted that the coupling must be as low as possible, the reasons is to ensure changes to one part of the system should have minimal impact on the rest of the system. But it is also true that no system can be devised with zero coupling using OO concepts because there are always some connections needed between system components (like classes and methods) which are required to exist for system to work. The strength of coupling is an important factor, which signifies that some forms of coupling is stronger than the others, this may be due to the number of connections or the nature of the connections.

There may be various forms of dependencies which may exist in OO software. A form of dependency that is greatly likely to impact maintenance is one that is complex in nature and is difficult to detect. The important thing is to understand that how exactly the presence of such dependencies could influence the tasks of understanding and modification. The problem with defining a complexity measure is that it is not sufficient to provide an arbitrary quantification of some aspect of the program. Instead there needs to be a sensible theory or model as to how certain

aspects affect understandability and modifiability, which could later be corroborated through empirical observation. This is part of a proper software measurement process.

According to [Booch et at. 1999], in OO design there are three types important relationships among the classes, namely dependency, inheritance and association relationships. Dependency relationship represents the using kind of relationships between the classes; inheritance relationship connects generalized classes to their specialized classes; and association relationship shows the structural relationship among the objects. According to Baig [2004] we can define various metrics to compute the measure of above relationships. He provided the following metrics in his research:

- Number of used classes by dependency relation (NUCD)
- Total number of evidences for 'Used classes by dependency relation' (TNUCD)
- Ratio of NUCD to TNUCD (RNUCD)
- Number of user classes for a class through dependency relation (NUCC)
- Total number of evidences for 'User classes through dependency relation' (TNUCC):
- Ratio of NUCC to TNUCC (RNUCC)

When the dependency relationship between classes is not directly evident from the code, but some indirect form of dependency exists between two classes such type of dependency is termed as Indirect dependency or Indirect coupling between classes. In such case changing code of one class does not creates any problem in compilation of the other class, but on execution the incorrect results are generated or Exceptions occurs. Such type of relation has been discussed in detail in the next chapter and metrics is proposed to measure them.

3.3.2 Cohesion Metrics

One of the important properties of OO software is abstraction at class level. The attribute which captures the quality of this abstraction is cohesion for the class under consideration. A high value of cohesion typically represents a good abstraction by the class. One measure of OO cohesion measure given by Chidamber and Kemerer [1991] represents inverse measure of cohesion which they called as Lack of Cohesion in methods (LCOM) and defined as the number of pairs of methods operating on disjoint sets of instance variables, reduced by the number of method pairs acting on at least one shared instance variable.

According to the IEEE Standard Terminology, "*cohesion is the degree to which the tasks performed by a single module are functionally related*." A class is said to exhibit a high degree of cohesion if the attributes and methods in that class exhibit a high degree of semantic relatedness. The high cohesion software development pattern suggests keeping the highest level of cohesion possible in OO classes. In other words, each element in the class shall be essential for that class to achieve its purpose.

Cohesion metrics can be defined on various criterions. For example one criterion can be connection types. Lee defines three cohesion metrics with different connection types among the components of a class (i.e., methods and attributes) in a class. cohAR(c) measures the number of attribute reference connections of a class c, cohMI(c) measures the number of method invocation connections of a class, and cohAS(c) measures the number of attribute sharing connections of a class [Lee 2007]. Table 3.1 shows the three cohesion metrics based on the connection type.

Table 3.1 Cohesion Metrics Based on the Connection Type

Symbol	***Connection Type***	***Element 1***	***Element 2***	***Description***
cohAR(C)	Attribute reference	Method *m* of class *c*	Attribute *a* of class *c*	Attribute reference: *m* references *a*
cohMI(C)	Method invocation	Method *m* of class *c*	Method *m′* of class*c*	Method invocation: *m* invokes *m′*
cohAS(C)	Attribute sharing	Method *m* of class *c*	Method $m'$of class $c, m \neq m'$	Attribute sharing: *m and m′* reference an attribute *a*

Another criterion for cohesion can be based on domains of measure and we can apply above metrics to class and class cluster domains. For example, aCohAR(s) can be defined as the averaged CohAR(c) of classes in some class cluster *C* and measures the averaged attribute reference cohesion metrics of classes in the system.

A third criterion based on Direct or indirect connections also can be applied for cohesion, which can only choose direct connection and measure the direct connection. Cohesion is the degree to which the methods and attributes in a class are related. The higher connectivity between methods and attributes means the higher cohesion, and a low cohesive class has been assigned many unrelated

responsibilities. Consequently, the low cohesive class is more difficult to understand and harder to maintain and reuse.

Therefore classes having a low cohesion value should be considered for refactoring. For example we can extract parts of the functionality to separate classes with clearly defined responsibilities. Let us consider a sample code (Figure 3.3) showing cohesion in a class and cohesion metric values obtained by the system. Class A has two methods $m1$ and $m2$, and method $m1$ makes a method invocation connection by invoking method $m2$, thus the system calculates a choMI metric value of one (cohMI(A) = 1). For the cohAS metric, methods $m1$ and $m2$ establish an attribute sharing connection by sharing an attributes y and z, thus cohAS cohesion metric value of the class is calculated (cohAS(A) =2).

Cohesion in a class	**Metric Value**

`public class A {`
`  int x,y,z;`
`  A(){`
`  x = 10;`
`  y = 15;`
`  z = 20;`
`  }`
`  public void m1(){`
`  m2();` **//cohMI**
`  x = x + 2;` **//cohAR**
`  y = z – 3;` **//cohAS**
`  }`
`  public void m2(){`
`  z = y – 3;` **//cohAS**
`  }`
`}`
	cohMI(A) = 1
	cohAS(A) = 2
	cohAR(A) = 1

Figure 3.3 Cohesion in a class and metric values

Coupling and Cohesion among the classes in OO software plays an important role in effort done on testing of class units. Classes having a low value of cohesion and tight coupling require more tests to be performed as compared to simple classes having high cohesion and loose coupling. Therefore writing effective test cases at unit level is an important issue. The next subsection discusses software testing and automatic test case generation techniques at class level.

3.4 Software Testing

One of the most used quality assessment methods for software is testing. While testing OO software there are two important steps. The first is to initialize it with a set of values and then second step to provide the test data. The first state takes the test object in a single test state for the software. These values can be any primitive values like an integer or complex values like an object. Once the object is initialized its methods under test are invoked with proper test data in second step. If methods take some other objects as parameters then they are also required to be initialized with appropriate values. We can use software specification to determine the validity of test cases. It is generally not possible to test all the states of an object as the number of states may be very large therefore some criteria is used to test some selected states only which may be of interest. There are various types of testing but broadly we can divide them as black box and white box testing. Each of them can be applied at the different levels of testing stages.

3.4.1 Levels of Testing

There are basically three levels of testing i.e. Unit Testing, Integration Testing and System Testing. Various types of testing come under these levels.

- **Unit Testing** is done to verify a single program or a section of a single program. According to Koomen and Pol [1999] a unit test is "*a test, executed by the developer in a laboratory environment that should demonstrate that the program meets the requirements set in the design specification*". Whittaker [2000] states that "*unit testing tests individual software components or a collection of components. Testers define the input domain for the units in question and ignore the rest of the system. Unit testing sometimes requires the construction of throwaway driver code and stubs and is often performed in a debugger*".

- **Integration Testing** is performed to verify interaction between system components. Prerequisite for this level is that unit testing must be completed on all components that compose a system. Integration test is the testing to ensure that interfaces to code external to the class being integrated with the other classes in the software are correct. Techniques used

are similar to unit testing and system test techniques can also be applied. Usually there is a nontrivial overlap between the unit and integration test activities.

In general, drivers and stubs are simplified versions of the units that, upon completion, will be used in the desired software system. The simplification allows one to establish a controlled environment and to test the units at an early stage. Simplification also limits unit testing as the use of drivers and stubs, in general, it fails to reproduce the complete environment where the unit being tested would be if it was interacting with the actual units in the software system.

- **System Testing** is done to verify and validate behaviors of the entire system against the original system objectives. System testing involves the complete system comprising software, hardware, databases and network resources. System testing is done after software has been created. Most people understand by testing as system testing only. It is the process of executing the software with the intent of finding errors [Myers 1979]. Basically it includes finding those errors which are related to software's external quality characteristics. System testing serves to compare the system to its original specifications. Original documentation is required for system testing to be completed. Usually System testing is performed at the end of the project that means that every defect found during system testing often requires more efforts.

3.4.2 Types of Testing

Software testing methods are traditionally divided into white box and black box testing. These two approaches are used to describe the point of view that a test engineer takes when designing test cases.

- **Functional testing** or black box testing is "testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions" [Gao et al. 2003]. Functional testing is directed at executing test cases on the functional requirements of software to determine if the results are acceptable. For functional testing equivalence class is a common technique. According to Jorgensen [2002]:

"*The use of equivalence classes as the basis for functional testing has two motivations: we would like to have the sense of complete testing, and at the same time, we would hope that we are avoiding redundancy*"

- **Structural testing** or white box testing is done to test the functionality of the program with test cases based on the structure and logic of the design and source code. Complete structural testing exercises the program's data structures (such as configuration tables) and its control and procedural logic at the different testing levels. There are two benefits of structural testing; the first is the creation of test cases based on the logic of the application. The second is the detection of how successful tests are by examining how many different paths through a program were executed. "In path testing, a major difficulty is that there are too many feasible paths in a program. Path-testing techniques use only structural information to derive a finite subset of those paths, and often it is very difficult to derive an effective subset of paths" [Gao et al. 2003].

  The use of test cases based on the logic of programs would require a graph of the nodes and connecting paths. Then some equivalent methodologies will be needed to determine about test cases to be created and metrics needed to decide how effective the test cases are. The tester will need to select the appropriate test data to cover each path at least once as per the coverage criteria. But it cannot guarantee to detect all the errors because the program logic may contain a large number of paths through the program. To increase the rate of error detection a number of metrics can be calculated to evaluate just how successful test cases are.

- **Hybrid testing** orgray boxtesting technique combines both structural and functional information to perform the tests [McMinn 2004]. It works like a Black Box test; however, the tester has a limited knowledge about the implementation details, or about the algorithm of the code. This type oftest iswidely usedin applications that requireserveras a database, and systems thathave databasesas a repositoryof information.

3.5 Analysis of Structural Testing

Each approach in testing has one or more criteria to check whetherthe test cases used are actually covering the requirements of the test in problem. This section points out some of the main criteria used by theWhite Box approach, which is described in this work.

3.5.1 Code Coverage Analysis

To measure how well the program has been exercised by a test suite, one or more coverage criteria can be used. Code coverage analysisis a criterion often used in structural testing, and consists of a process that covers three main activities:

1. Find areas of a program not exercised by a set of test cases.
2. Create test cases to increase coverage, and
3. Provide a quantitative measure of code coverage, which consists of an indirect measure of quality [Cornett 2002].

Analysis of coverage does not guarantee the quality of software being developed, but it is about the quality of test set sthat are testing the software. This technique is useful in evaluating the results of unit tests, which directly analyze the behavior of the code. There are a number of coverage criteria, among those Statement coverage, Branch Coverage, Condition Coverage, Path coverage are main.

The complexity of the logic is determined by the number of different nodes and the number of different possible paths through the program code. Using the above metrics it enables a tester to determine the extent to which the code has been executed. Statement coverage measures the number of unique statements which have been executed. The main advantage of statement coverage is that it can be measured directly from program code. But it is too simplistic and usually not a good testing evaluation criteria [Silva and Someren 2010].

Branch coverage measures the unique evaluation of Boolean expression from conditional statements. It is simple to compute and it is stronger than statement coverage. Full branch coverage implies full statement coverage. However, it is insensitive to complex Boolean expression and it does not take into account the sequence of statements. Similarly, condition coverage measures the unique evaluation of each atomic Boolean expression independently of others. It provides a more sensitive

analysis compared to branch coverage. Full condition coverage does not imply full branch coverage since branches might exist that is unreachable. Another criteria using path coverage measures unique paths a program execution can take. It requires thorough testing to achieve a good path coverage, it is very expensive, and in many situation unfeasible, since the number of paths is exponential to the number of branches and there are paths that are impossible to execute. For instance in Figure 3.4, it is not possible to execute a program that has S_2 and S_6 on its path.

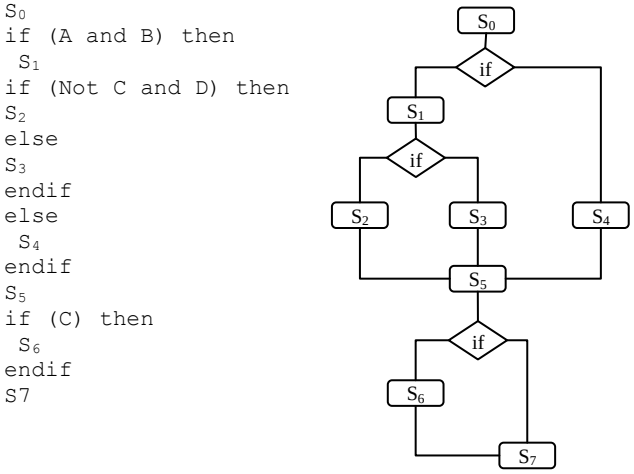


Figure 3.4 Example of Program Flow Analysis

3.5.2 Automatic Generation of Test Data

Automated testing plays a great role in software development due to the efficiency and accuracy of the executed output. It gives the software developers the assurance of producing reliable and quality software that is easy to use and understand by the end users. Themanual wayis the mostlyused, but it has the disadvantage of being expensive (long time of preparation of the data, long time to run the software). The best alternative is then automatically generating the data. Automatic generationof data canbe randomor throughcertaintechniques createdto find agood set oftest data.It is consideredagood setof dataas thatwhich coversthe criteria forthe test in problem, how to achieveahighcode coverage. Some of the techniquesused are: alternating variable method [Ferguson and Korel1996], iterative relaxation method [Gupta et al. 1998], simulated annealing [Tracey et al. 1998], genetic algorithms [Michael and McGraw 1998], rule-based [Deason et al. 1991] etc. Genetic Algorithms (GAs) are discussed in detail in section 3.6.1.

In the following section we will discuss optimization techniques based on evolutionary search which transforms the test data generation into an optimization problem.

3.6 Evolutionary Algorithms

Evolutionary Algorithms (EAs) are search techniques based on meta-heuristics theory of survival of the fittest, Charles Darwin. These techniques simulate the evolution as a search strategy to generate candidate solutions using operators inspired by genetics and natural selection [McMinn 2005]. EAs can be effective in finding local maxima of complex andn on-continuous problems that are very difficult to resolve [Rela 2004]. Among the methods which compose the group of EAs, we can name: genetic algorithms, Evolution Strategies, Cultural Algorithms and Genetic Programming [Mantero and Alander 2005].

GAs are the best known form of EAs, and were initially studied and presented by Holland [1975]. GAs are explained in more detail in the next subsection. Genetic Programming (GP) is a machine learning technique used to optimize a population of programs in accordance with an objective function, which is based on a problem that the programs require candidates to resolve. Evolution Strategies (ES) work with vectors of real numbers as a representation of the solutions, and uses selection and mutation as operators. Mutation rates are usually self-adaptive.

3.6.1 Genetic Algorithms

GAs are search and optimization techniques inspired by evolution, the best known EA [Whitley 1993]. These algorithms are based on the principle of natural selection and survival of the fittest. Figure 3.5 shows a typical GA. GAs work with populations where each individual's population corresponds to a solution in search space is represented by a chromosome. The chromosome is usually represented by a string of bits that are part of the solution of optimization problem in question. The representation of a solution in the form of a chromosome is dependent on the problem.

The first step in a GA, as shown in Figure3.5, is to generate an initial population. This generation is usually random, unless there is prior knowledge of the search space. Then the initial population of each chromosome is evaluated according to an objective function, and new populations are evolved iteratively, which involves following steps:

(1) Selection: This involves selectingthe best individualsof the current population, i.e., those withthe highest value ofthe objective function.In the nomenclature of GAs, the objective function is usually called the fitness function (or fitness).The individuals selected are stored in an intermediate population.

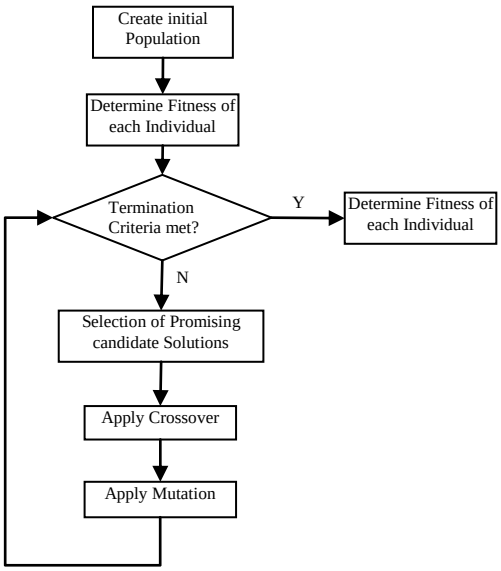


Figure 3.5 Flow of a Typical Genetic Algorithm

(2) Crossover: a new population is generated from the crossing over the fittest individuals selected in the previous step. In crossover operation, a pair of individuals in the intermediate population is selected at a time and their chromosomes are combined. This combination is made, in general, choosing a random point of separation along the length of each chromosome. Figure 3.6 illustrates the use of this operator. In the figure we can observe that each parent chromosome (represented by bit strings) is cut at a certain point. Then the first part of parent-1 chromosome is concatenated to the second part of parent-2 chromosome and the first part of parent-2 chromosome is concatenated to the second part of parent-1 chromosome, generating two new child chromosomes [Duda et al. 2000]. The crossover is applied with agiven probability for each chromosome pair; this probability is called the

crossover rate. When there is no crossover, the children are the same as parents and certain features are preserved with it. The crossover operator is applied successively for different pairs of individuals selected until a complete new population of individuals is generated.

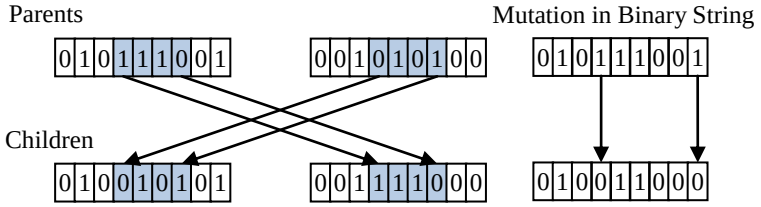


Figure 3.6 Crossover

Figure 3.7 Mutation

(3) Mutation: here each bit in a chromosome has a small chance of being changed from 1 to 0, or vice versa [Duda 2000]. This operation aims to increasethe diversity of solutions generatedfrom one population to another.The mutation operatoris also applied with a given probability, the mutation rate. To avoid too abrupt change from one population to another, it is recommended the use of small mutation rates [Lacerda and Carvalho 1999]. Figure3.7 illustrates the use of this operator, where some bits of the child chromosomes are modified as the mutation rate.

(4) Re-evaluation: The chromosomesof thenew populations are re-evaluated according to thefitness function, so that anew population isgenerated fromoperatorsmentioned above, until astop conditionis satisfied.

The basic idea of GA is that new points of search are defined through a combination of successful solutions of past populations, as occurs in nature.The rate of mutation is to explore new regions of search space and avoid premature convergence. Other operators can be also used as Elitism [Dejong 1975], which is to keep the best chromosome from one generation to another while using the algorithm, with the intention of preserving the best solution found so far.

3.7 Evolutionary Testing

Optimizing search techniques such as EAs are used by Evolutionary Testing (ET) to generate test data. The search space becomes the input domain of the test object, where each individual or potential solution is encoded set of inputs to that test object. The fitness function is designed to find test data for the type of test that is being undertaken. As mentioned earlier, testing is a critical step in software quality. Yet it is an expensive process to consume much of the cost and effort involved in

software development. Thus, the useof techniques and strategiesto reduce costsassociated with each phase of testing, as well as thetime spent is of great importance and relevance for the ET.
Software Engineering (SE) as awhole contains a variety of problems that have a lot of possible solutions. To find the ideal solution in many situations it is theoretically impossible or intractable in practice [Harman and Jones2001]. This problem needs to use the optimization techniques and seeks trong candidates to find solutions to the problems of the SE. Software Testing can be seen as one of the problems that need SE solutions for its techniques to be implemented and its phases to be executed in the best possible way. There are some ways to streamline the process of Software Testing, without lack of quality. One of these forms is the automation or semi-automation of tests, which transfers the efforts manual part of building test cases of developers and testers for using their expertise in developing the automation tools that would work for them.

The automatic generation of test data usingEAs, also called the Evolutionary Test has given an important contribution to the automation process of test generation [Baresel etal2002]. The technique consists in generating input data to cover certain structural or functional criteria of a program, using EAs to perform a search in the space of possible inputs of the system.

As we discussed earlier that testing an OO software class requires broadly two stages, first is to bring it into required test state. This will require certain method call sequences and constructor calls to be generated. There can be various call sequences in general. We have to generate only those call sequences which are of interest as per some defined criteria. In second stage of testing we need to pass appropriate parameters to Method Under Test (MUT) and perform structural testing of that method according to some coverage criteria. We use GAs for both of these cases which are discussed in next subsections.

3.7.1 Structural Evolutionary Testing for Methods

The process of generating relevant test cases for a given software unit is considered to be the most important task in software engineering area. If performed manually, it is time consuming and error-prone, therefore it becomes very costly. Evolutionary structural testing [Baresel et al. 2002] is an approach to automate the process of test case generation. The task of generating the appropriate input

data which may lead to the execution of a particular program element is formulated as an optimization problem which is tried to be solved using an EA.

An EA is a meta-heuristic optimization technique that is inspired from the principles of the Darwinian theory of biological evolution. The workflow of a simple EA has been described in Figure 3.5. At the initial level, a set of candidate solutions for the given optimization problem has to be generated randomly. This initial set is called population it is further modified iteratively in order to find an ideal solution. As the next step, this candidate solution is evaluated by use of a fitness function. The fitness function is used to assign each candidate solution a quantitative value which is according to the ability of the candidate solution to solve the optimization problem.

The assigned fitness value determines the creation of new individuals, therefore the fitness function becomes the most important part of EA and it also has a crucial role in the success of the optimization. After evaluating the candidate solutions, if ideal solution is not found then two modification steps are taken which are known as crossover and mutation operations. To perform crossover, the two eligible candidate solutions are chosen according to their fitness value. A candidate solution associated with a better fitness value has more probability to be selected for crossover. The crossover operation over two candidate solutions produces two offspring candidate solutions which have similarity with their parents and have a better ability to solve the optimization problem. Then mutation operation is performed over candidate solutions which mean that some of their part is randomly changed. After these two operations the candidate solution is again evaluated for fitness using fitness function. This cycle of crossover, mutation, and evaluation, is repeated until some termination criterion, such as the optimized solution is found or resources are exhausted.

When EA is applied to structural testing, it aims to generate test cases that cover a particular program element. Typically these program elements are statements or branches of the program under test, which further depends upon the selected test adequacy criterion. For example, if a test case generator is expected to provide a set of test cases which can give high branch coverage then an individual evolutionary search for each program branch needs to be performed. The complete set of test cases generated by EA provides the final test suite.

Whenever some test goal is defined by the tester it requires the definition of an individual fitness function. The fitness function is based on the distance of the execution flow which is produced by a candidate solution with that of the targeted program element. There are two popular metrics for defining this distance: approximation level and branch distance [McMinn 2003]. An approximation level is defined to measure the number of correct branches taken to reach a desired program construct. It is defined in terms of the control flow graph of the program under test. Solutions with higher approximation levels get better fitness values. Approximation level measures the number of potential problem nodes of the shortest path from the problem node to the targeted program element where the problem node is that node of the control flow graph at which execution diverged down a branch (the critical branch) which makes it impossible to ever reach the target.

Branch distance is concerned with the condition assigned at the concerned node to express the closeness of execution in taking the other branch and avoiding critical branch. For each relational operator occurring in a condition, a particular distance function will be applied [McMinn 2003]. For instance, in case of a condition if $(a == 200)$ the distance function for the equality operator is defined as $d = |a - 200|$, mapped into the range [0, 1]. The evolutionary search tries to minimize the approximation level and branch distance. In this case 0 indicates that the candidate test case has covered the test goal of interest.

Measurement of structural coverage [Weiser et al. 1985] of code is a means of assessing the thoroughness of testing. Coverage metric is expressed in terms of a ratio of the paths executed or evaluated at least once to the total number of paths. This is usually expressed as a percentage.

$$Coverage = \frac{paths\ executes\ at\ least\ once}{total\ number\ of\ paths}$$

The aim of applying evolutionary testing to structural testing is the generation of a quantity of test data, leading to the best possible coverage of the respective structural test criterion. The structural test criteria are divided into four categories, depending on control-flow graph and required test purpose [Sthamer et al. 2002]:

- node-oriented methods
- path-oriented methods
- node-path-oriented methods, and

- node-node-oriented methods.

Tests are separated into partial aims and fitness function is defined for completion of partial aims. For achieving a preferably large coverage of the selected structural test criterion, each partial aim needs to be executed, as an example all test object statements are passed through to achieve a higher degree of statement coverage. Therefore, for the evolutionary test, divide the test into partial aims that result from each of the specified structural test criterion. Control-flow graph of the program under test is used to define the partial aims.

3.7.2 The State Problem

Object oriented software components can store internal data, and can exhibit different behaviours based on the state of that data. The existence of state behaviour in test objects [McMinn 2005] presents new challenges for evolutionary test data generation. Certain strategies may require the generation of input sequences. States in test objects present two major challenges for evolutionary structural test data generation. The standard evolutionary approach generates input vectors for single function calls. Test objects with states may require a sequence of method calls to be generated in order for certain structures to be covered. This sequence may include calls to several different functions.

3.7.3 Evolutionary Testing to Generate Method Call Sequences

For unit testing of OO software using search based techniques has not been investigated comprehensively. There are mainly three approaches available in literature [Wappler and Wegener 2006a]. The test case generation problem is formulated as a search problem and then this problem is tried to solve using EAs. In unit testing of OO software it requires to create test cases which have a method call sequence which realizes some test scenario. These test cases cannot be used directly to be applied for EAs. They are required to undergo some representation which is compatible for EAs to be applied. Since the search space for the test programs is large and they are required to evolve as a better solution a fitness function is used to guide them during the search process.

Strongly Typed Genetic Programming (STGP) is an approach [Montana 1995] which is used to generate the OO test programs. In this approach method call sequences are represented by method

call trees. The call dependencies of the methods which are relevant for a given test object can be expressed using these trees. Call dependency in this representation is preserved while applying evolutionary operators when they work on the method call trees. This approach can be applied to generate test cases which satisfy branch coverage of Java classes. Not only that, it can be easily adapted to work for other coverage criteria also.

3.8 Genetic Algorithms for Class Testing

The complexity of objects is increased in ET representation, therefore to deal with this enhanced complexity of objects the chromosome representations must be enriched which are capable to deal with these more complex entities [Tonella 2004]. Some grammar is needed which can add structure to the chromosome during evolution that can be mapped directly to an executing program. In [Tonella 2004] various notations are introduced which can work as a base for structuring the chromosomes while applying genetic approach to OO software.

A random initial population of chromosomes is chosen for the GAs to start with. Selection process is used to choose the chromosomes to be recombined and then these are mutated out of this initial population. The selected individuals in recombination reproduce other individuals by exchanging their information in pair-wise manner. This exchange of information is called crossover. A small change to each selected chromosome is applied uring mutation process. The resulting chromosomes are then evaluated through a fitness function. This transfer the information encoded in the chromosome, the so-called genotype, into an execution of the Software Under Test (SUT), the so-called phenotype. The fitness function measures how well the chromosome satisfies the test criterion. In this case it is the coverage of test code. The implementation of the fitness function follows earlier standards in ET, described in other articles, i.e., [McMinn 2004, Gross and Mayer 2002, Tonella 2004].

3.8.1 Chromosome

The GA is required to generate test programs, but since they don't have understanding of programs, statement or objects therefore some kind of encoding of the GA components is required to be defined which allows this representation of a test program as a chromosome. These chromosomes can be used with the GA. The chromosomes in GP represent hierarchically structured computer programs

made up of arithmetic operations mathematical functions, boolean and conditional operations, and terminal symbols, such as types, numbers, and strings. The fact that GP is based on hierarchically organized trees requires specialized genetic operators for crossover and mutation [Gross and Mayer 2002]. In presented research test case sequence is represented using trees. The information of methods which should be called in sequence is encoded in these trees. It also contains the information about target objects and parameter objects which should be used for the individual method calls. Every operation refers to some object, which is associated with some type. Some input parameter values as well as their type is also associated with the operations. These must be created by the GP process and added as leaves to the nodes in the tree representation of the test cases. A sub tree is used to map each statement is to the entire GP hierarchy, which includes constructors and input values for the required object. Besides the user defined object types, the basic primitive types such as boolean, integer, real etc. must also be permitted. These object types and the basic primitive types are used primarily to denote input and return values.

For example consider the Figure 3.8 which shows an example test cluster for class B which is considered Class Under Test (CUT) here. The test cluster consists of class A and class B.

```
class A
{
	private int x;
	public A(int x){this.x=x;}
	public int getx(){return x;}
	public void setx(int x){this.x=x;}
}
class B
{
	private int y;
	public B(A a){y=a.getx()*a.getx();}
	public void test(A a,Boolean b)
	{
		if(a.getx()>10)
		System.out.println("Greater than 10");
		else
		System.out.println("Less than 10");
		if(b==true)
		System.out.println("true");
		else
		System.out.println("false");
	}
}
```

Figure 3.8 A sample test Class Cluster

The objects of class *A* are used as parameter for the constructor of *B* and for method *test* and therefore they are required when we test CUT *test*. Figure 3.9 shows an example tree shaped representation of a GP chromosome that translates into the following test case:

```
A a1 = new A(3);
A a2 = new A(11);
B b = new B(a1);
b.test(a1, false);
```

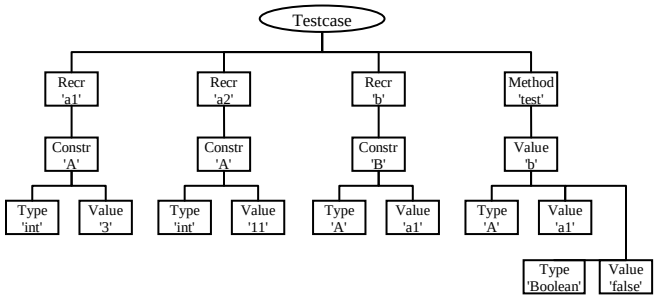


Figure 3.9 Tree Representation of Chromosome

The test scenario consists of the creation of two instances of class A and one instance of class B where one instance of A is used as the parameter object. Then *test(A, Boolean)* is called with second instance of A and a Boolean parameter. We are interested in those test cases which may cover all the paths in method *test*.

3.8.2 Constructing Initial Population

The first initial population may either be created randomly or it may be a population based on execution traces. The initial population which is created randomly contains randomly selected input values as well as the initial method invocations are also random. If execution traces are used for generating the population then it uses the existing knowledge for executing the SUT. In this manner an initial population is generated which can already cover many of the SUT's runtime paths for a typical usage profiles. The performance of test generation is significantly increased by using this method.

3.8.3 Mutation and Crossover

Mutation is a genetic operator, when applied it alters one or more gene values in a chromosome from its initial state. This can result in entirely new gene values to be added to the gene pool. GA may be able to arrive at better solution with these new gene values, than was previously possible. Mutation is

an important part of the genetic search as it helps to prevent the population from stagnating at any local optima [Piszcz and Soule 2006]. A separate mutation operator is required in GP, each of which may be subjected to mutation according to a predefined mutation rate. Three types of mutation operators can be identified for OO software, the first type of operator is used to create a new building block, second type is that makes changes to an existing one and the third is used to delete an existing building block. These three operators are are generated and used for each of the components of the test case statement. Applying these operators to a constructor, it can be created, deleted or changed to a different constructor. Similarily, by applying to the normal methods they can also be created or deleted. By applying these operators to method parameters their value can be altered. Whenever a method or constructor is created or deleted, it constructs or deletes its subtree and input parameters also. Constructor is the method which needs to be called for creating any object before any of its methods are called. Figure 3.10 and 3.11 show the process of mutation when applied to a tree representation.

The crossover is applied in GP at the nodes which represent genes of the chromosome tree. The nodes over which crossover is to be applied can be determined randomly, if they are found compatible then the crossover operator can be applied and as a result of this operation their subtrees are exchanged. The compatibility of two nodes is same if they have same type of root node. The simplest way of crossover is the case where the node representing entire method including its input parameters in exchanged. The node comprising of input parameter genes may also be exchanged if their type matches. When crossover and mutation are applied many times they can generate chromosomes of arbitrary length, just by simply adding more and more trees. This overgrow of chromosome tree is undesirable and can be regulated by the introduction of penalty on the overall fitness for larger individuals.

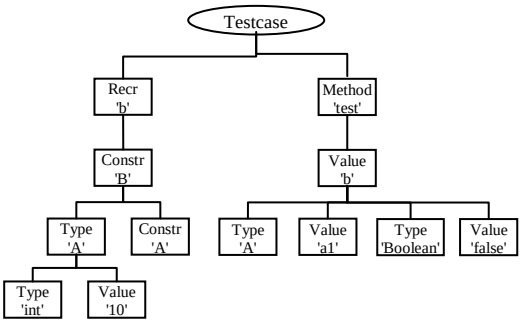


Figure 3.10 Tree Before Applying Mutation

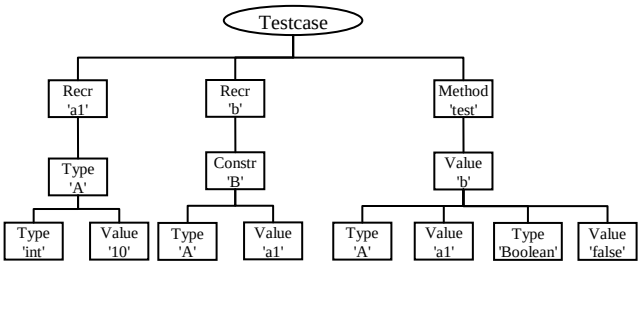


Figure 3.11 Tree After Applying Mutation

3.9 Summary

For a programmer it is important to identify reusable and maintainable classes in OO software. In this chapter we discussed metrics which are extensively studied and statistically analyzed to show internal characteristics from classes in OO software. These metrics also define the complexity of a class. The more complex a class is, it will need more effort during testing. There are different ways to lowerthe testing effort by automation of certain phases of the test with the automatic generation of test data or programs. Such techniques are widely used in recent decades. We also discussed meta-heuristics techniques that are dynamic and can be adapted to different test objectives. The algorithms in the Hill Climbing and simulated Annealing were briefly discussed and EAs have been presented as the most effective framework to accomplish this form of automated testing, especially GAs. The principal technique was exposed in ET, which aims to unite some EA and data generation for testing. Using this technique alone an automation or semi-automation of the testing process can be achieved and the results can be very satisfactory as confirmed in several previous works.

In the next chapter we discuss the research methodology which has been followed in this thesis.

CHAPTER 4 *Research Methods and Experimental Design*

This chapter discusses the different measurement and testing methodologies which are used in this thesis which can help to improve the quality of the object oriented software.

4.1 Software Quality Measurement

One important question that arises in software engineering is how software quality can be measured. What, where and when we assess and assure quality, are still open issues. Many views have been expressed about software quality attributes including maintainability, evolvability, portability, robustness, reliability, usability, and efficiency. During software development process testing has always been a vital part, but thorough testing of software is a cumbersome job for a tester. Therefore the test cases are generated according to some test adequacy criteria. This thesis is about establishing a relationship between indirect coupling and maintainability, because accurately measuring the former enables the prediction of the latter. Also, an approach which may guide the search into promising, unevaluated areas of the search space has been proposed, thus generating improved quality of software test cases.

Indirect coupling has an important role in maintainability of the software. Therefore the exact nature of this form of coupling is important to understand. The quality of developed software is tested through software testing activities, for which various testing techniques have been developed in past. This chapter discusses the methodology used in this research to relate indirect coupling with software maintainability and also the methodology used to improve the fitness of feasible as well as unfeasible test cases in automatic generation of test cases. This chapter also provides the methodology used for data analysis and its validation.

4.1.1 Indirect Coupling and Software Maintainability

The following methodology is chosen, consistent with the philosophy of software measurement and testing presented by key researchers [Baker et al. 1990; Myers 1979; Fenton 1991]. Essentially the methodology comprises the following key steps:

1. Define the attribute to measure (indirect coupling).
2. Develop a theory or hypothesis on a causal relationship between the attribute and another important quality attribute (maintainability).
3. Empirically confirm the theory or hypothesis. The results may suggest a refinement to the theory to more accurately reflect the reality, or lead to the discovery of other interesting related factors that affect maintainability. In either case the process will be repeated until a satisfactory, empirically validated theory is achieved.

The success of the above methodology lies in doing proper measurement, i.e. defining attributes on an unambiguous, theoretically sound basis and making any assumptions about the measurement explicit. While we are ultimately interested in being able to predict tangible external attributes such as time and cost, it cannot be done without accurate measurements of the relevant product attribute. To put it in the context of our research, ensuring accurate measurement of indirect coupling (step 1 above) must precede any empirical validation effort.

There is a common misconception that a measure is only valid (hence worth studying) after it is shown to be statistically related to some data pertaining to an external software attribute [Baker et al. 1990]. Indeed there are many studies (discussed in chapter 2) that attempt to establish predictive capabilities of internal metrics such as coupling to external attributes such as fault-proneness by statistical regression tests. Such models focus on establishing a posteriori relationships between the internal measure and external measures [Melton 1996]. Without a thorough understanding of the exact reason for such correlations, we would very likely end up misusing our metrics. If the measure is not properly defined and there are no specific grounds as to expect the measure to be correlated with the data, then that correlation would not be saying anything meaningful [Fenton 1994; Baker et al. 1990].

In order to be able to predict, we need a priori theories as to why such a relationship would occur in the first place, which can then be empirically confirmed or falsified and refined until a satisfactory causative relationship is established [Baker et al. 1990]. Of course this is not a trivial process, but the importance in investigating exactly what it is about the internal attributes that would give effect to external attributes is significant. We want to have a clear model of how the attributes are mutually related so that we can better understand and predict them.

4.1.2 Automated Test Data Generation

In the presented strategy class specification is used to obtain class state space, which is partitioned into substates. A test model is to be developed which is composed of a set of states and a set of transitions among the states. State space partitioning of the class is required to perform and each state is obtained by using it. Each transition consists of a method, which can change the value of an object from source state to target state. The input space of each method is the sets of values for the input parameters of the method. This input space is required to be partitioned. The input space partition values can be used with test model to obtain the test data. Finally this test data can be used for the generation of test cases.

We apply genetic algorithms, which is a kind of evolutionary algorithm. The basic idea of a genetic algorithm is to start with a randomly initialized population of individuals. Each individual is a potential candidate solution of a given problem. A fitness function is used to evaluate the adequacy and quality of each individual. After this, a selection process, which is based on the fitness associated to each individual, extracts a subset from the current population. This means that fitter solutions are more likely to be selected. These selected individuals are combined to form a new generation of population. The combination is usually done through a crossover operation, which takes two individuals and exchanges their information at a random selected position. Often a mutation process is applied, to prevent that individuals become too similar and thus the population is evaluated again, and the process is repeated until a specific termination condition is satisfied.

Unlike procedural programs, in object oriented software an object encapsulates a state and its behavior at runtime, depends not only on arguments, it has received but also on its current state. With our evaluation methodology, the quality of a particular Test Program has been related to the Control

Flow Graph (CFG) nodes of the method which are the targets of the evolutionary search at the current stage of the search process; Test Programs that exercise less explored (or unexplored) CFG nodes and paths must be favored by this approach.

Control Flow Graph: A control flow graph is a flow graph representing the possible control flow of a program and taking into account the label of each statement. So, a node in the control flow graph corresponds to a statement with its label in the program and edges represent the possible transfer of control flow between statements. The difference with the flow graph is that for the control flow graph we consider only the possible paths regarding inputs of the program. We define the Label function in order to associate each node to each label.

A control flow graph describes the sequence in which the different instructions of a program get executed. In other words, a control flow graph describes how the control flows through the program. In order to draw the control flow graph of a program, all the statements of a program must be numbered first. The different numbered statements serve as nodes of the control flow graph. An edge from one node to another node exists if the execution of the statement representing the first node can result in the transfer of control to the other node. In order to understand the path coverage-based testing strategy, it is very much necessary to understand the CFG of a program.

Linearly independent path: A linearly independent path is any path through the program that introduces at least one new edge that is not included in any other linearly independent paths. If a path has one new node compared to all other linearly independent paths, then the path is also linearly independent. This because, any path having a new node automatically implies that it has a new edge. Thus, a path that is subpath of another path is not considered to be a linearly independent path.

4.1.3 Coverage

In the field of software testing, coverage of source code is a measurement of how much code has been explored through executing test cases. This measurement is normally given in percentages. There are many different coverage criteria.

Code Coverage: The degree to which the source code under test has been tested defines the code coverage. There are different ways of defining coverage and different interpretations will yield distinct results.

Path Coverage: This research uses path coverage as a measure for test case effectiveness. It is suggested that path coverage should be sufficiently fine-grained that reductions in test case effectiveness will likely introduce significant decreases in path coverage of the SUT. Path coverage generally acts as a test for regressions, checking for a decrease in quality of the test cases, rather than a method for exposing faults that have been present in different runs of test cases. We discuss below the possibility of using path coverage results to suggest "improvements" to the test cases. The path coverage-based testing strategy requires us to design test cases such that all linearly independent paths in the program are executed at least once. A linearly independent path can be defined in terms of the CFG of a program.

Measuring Path Coverage: We use our custom tool to instrument the SUT in order to record path coverage information. In our approach while measuring path coverage it is considered to be a simple end-to-end measure, for SUTs in which test cases consist of an input value which produces an execution and output. However, for the stateful systems that are perhaps most suitable for random testing and model-driven verification, a test sequence typically consists of a series of function calls. We choose to measure path coverage at the granularity of top-level function calls. That is, we maintain a set of paths covered for each top-level-entry function that is called by the test harness. A separate set of path record is maintained for each test execution of the called method. Path is recorded as a bit vector, containing every if-then decision made by the execution of the function, from entry until return to the test harness. This bit vector includes all decisions made in functions called by the top-level function, recursively, and therefore records all path information.

In general the number we will use to measure tester effectiveness is simply the total number of unique paths through functions executed during a test run.

4.1.4 Coverage measures

Because complete paths cover a certain number of nodes or edges, it's possible to define two percentages, representing the node coverage and edge coverage of a set of paths.

Definition: Let Π be a set of complete paths in the graph *G* having set of nodes *N*, set of edgs *E* and set of initial states n_1 represented by *G* (*N*, *E*, n_1). The Node coverage of Π is the percentage of different nodes that is covered by at least one path in Π:

$$NodeCov(G, \Pi) = \frac{N(\Pi)}{N} \times 100\% \tag{4.1}$$

Edge coverageis the percentage of different edges that is covered by at least one path in Π:

$$EdgeCov(G, \Pi) = \frac{E(\Pi)}{E} \times 100\% \tag{4.2}$$

In a graph, many different complete paths exist. Two percentages called path coverage and simple path coverage can be defined.

4.1.5 Data Analysis and validity

The experimental data obtained will be represented using tables. The data for maintenance and indirect coupling must be correlated for the validity of the model. This needs a correlation analysis of the data obtained. The correlation analysis is conducted to examine the relationship (correlation) between, in this case, software maintenance and indirect coupling. The correlation between two sets of data can simply be checked by just drawing a scatter plot and the relationship between the two sets can be determined by visual inspection. There exist other more rigorous approaches also; one of these methods uses the statistical methods. This approach can be performed in two ways: by generating measures of association which indicates the similarity of two data sets. The second way is by generating an equation which describes the relationship between two data sets. In the presented analysis the scatter plot and the generating measures of association approaches will be used. There exist different ways which can be used to calculate the measure of association. The Spearman Rank Correlation Coefficient (SRCC) is a good method to use for software metrics, since it has ability to handle non-normally distributed data [Richard 1999]. If a correlation calculated by using this method results in a linear correlation then it indicates that the two sets of data follow each other in a

linear manner. Whenever using the scatter plot approach the two data sets are drawn in the same diagram with one set at the x-axis and the other at the y-axis. If there is a clear linear trend or pattern between the two sets it appears visually.

Spearman's rank correlation coefficient (SRCC): The value of SRCC always lies between between 1 and –1. A value which is nearby to 1 indicates a strong positive correlation and values lying near –1 indicate a strong negative correlation. Values lying near zero indicate weak or no correlation. For computing the SRCC (R) following equation is used:

$$R = 1 - \frac{6 \sum d^2}{n^3 - n} \tag{4.3}$$

The above equation contains *d* and *n* as the only unknown values. To get *n* just count the number of paired items in the two sets. To get *d* the two data sets must be ranked separately in either ascending or descending order (from 1 to n or from n to 1) [Richard 1999].

4.2 Summary

This chapter looked at the research methodologies used in this research. Justifications on why the researcher chose to use those methodologies were given. Empirical study has been followed to gather relevant data to achieve the research objectives. Experiments are conducted on small and large size Java open source projects and over standard Java libraries. The results are analyzed using correlations and graphical techniques.

The next chapter discusses indirect coupling and maintainability of the OO software. Then we propose our metrics which can relate indirect coupling and maintenance effort in OO software.

CHAPTER 5 *A New Metric for Software Maintenance in presence of Indirect Coupling*

One notion of quality in software design is how easy or difficult it is for the design to cope with change. A concept that is widely associated with quality is coupling, which is a "*measure of the strength of association established by a connection from one module to another*" [Stevens et al. 1974]. Several studies have identified clear empirical relationships between class-level coupling and class fault-proneness [Arisholm 2002, Baig 2004; Gui and Scott 2006; Offutt et al. 2008]. Previous research has shown that complex coupling relationships among OO software classes are among the critical factors that make testing and maintenance difficult and costly [Kung et al. 1995]. Therefore, analyzing and measuring software class relationship has gained increasing importance [Fenton and Pfleeger 1998; Li and Henry 1993]. In this chapter a metric and an algorithm is proposed to estimate the maintenance effort for software with indirect coupling between classes.

After the release of a software product the maintenance phase keeps the software up to date with the environment changes and user requirements [Erdil 2003]. The earlier phases particularly the design phase must be completed in such a manner that the product could be easily maintained. One major concern which affects a good object oriented software design is coupling between classes. Coupling is described as the degree to which one class is dependent on the other class [Briand et al. 1999]. If a class is having high coupling value then it is highly interdependent on other classes and vice versa. If a class is highly interdependent then any change in the class requires significant changes in other classes to which it is coupled [Weisfeld 2000]. Hence highly coupled classes require high maintenance effort. It can be noted that a system cannot completely be devoid of coupling for the proper functioning of software. There is some need of some connections among various sub classes of software. Hence maintaining loose coupling among classes is desirable characteristics of good software design [Hitz and Montazeri 1995].

Among the existing coupling measures, very few measures investigate indirect coupling [Yang and Tempero 2007b] connections. Most of the research has been applied only to direct coupling that is, coupling between modules that have some direct relationship. However little investigation has been

done into indirect coupling which can be described as coupling between modules that have no direct relationship. It is considered that indirect coupling is little more than the transitive closure of direct coupling which takes the path of the data flow between classes also into account.

In this chapter a metric based on this indirect coupling has been proposed. The proposed metrics are modifications of established metrics. The established metrics will be discussed along with a unified way of representing these metrics. There is also a description of exactly which aspects of the system they attempt to measure. It is explored that indirect coupling minimization defined by considering coupling paths we are able to relate indirect coupling to maintenance effort [Gupta and Rohil 2012]. This gives us a clear idea, how indirect coupling affects maintenance effort.

5.1 Levels of Coupling

Coupling is defined by the state of an object (state is represented as the value of its attributes at a given moment at run-time), and the state of an object's implementation (class interface and body at a given time in the development cycle) by Hitz and Montazeri [1995]. From these definitions, the authors derive two "levels" of coupling:

- Class level coupling (CLC): During the development lifecycle, the coupling which results from the state dependencies between two classes in a system is termed as Class Level Coupling.
- Object level coupling (OLC): During the run-time of a system, the coupling which results from the state dependencies between two objects in a system is termed as Object Level Coupling.

According to Hitz and Montazeri [1995], CLC is important when considering maintenance and change dependencies because changes in one class may lead to changes in other classes which use it. For these levels of couplings the authors have identified some factors which determine the strength of coupling. In CLC if a method of a class A invokes a method or references or an attribute of another class B then the following factors determine the strength of CLC between A and B

Type of access

- "Access to interface": A invokes a method of B.
- "Access to implementation": A references an attribute of B

5.1.1 Coupling Constitution Mechanisms

Table 5.1 presents the mechanisms that constitute coupling between classes in some frameworks [Briand et al. 1999]. One mechanism is represented in each row; an "X" indicates that the mechanism is covered by the framework in the respective column. For reference purpose numbers are assigned to the mechanisms.

Table 5.1 Mechanisms that constitute coupling

No.	Mechanism	Eder et al. 1994	Hitz and Montazeri 1995	Briand et al. 1997
1	methods share data (public attributes etc.)	X		
2	method references attribute		X	
3	method invokes method	X	X	X
4	method receives pointer to method			X
5	class is type of a class' attribute (aggregation)	X	X	X
6	class is type of a method's parameter or return type	X	X	X
7	class is type of a method's local variable	X	X	
8	class is type of a parameter of a method	X		
9	invoked from within another method	X	X	

5.1.2 Direction of Coupling

The framework given by by Briand et al. [1997] explicitly distinguishes between import and export coupling. Let us consider the two classes C and D being coupled through one of the mechanisms mentioned above. We may consider it as a form of client-server-relationship between the classes: the client class uses (imports services) and the server class is being used (exports services). Such a distinction between these classes has its importance. It may be difficult to reuse a class which mainly imports services in another context because it depends on many other classes. On the other hand, for a class which mainly exports services, the defects are particularly critical as they may propagate more easily to other parts of the system and are more difficult to isolate. We can make the conclusion that the direction of coupling measured directly influences the possible goals of measurement.

5.1.3 Direct and Indirect Coupling

Eder et al. [1994] used the transitive closure of direct interaction relationships to derive "indirect interaction relationships between methods" from "direct interaction relationships". The same idea can be applied to all kinds of coupling. That means, if a class C_1 uses a class C_2, which in turn uses another class C_3, then class C_1 is said to be indirectly coupled with class C_3. Therefore in such a case any defect or modification made in class C_3 may not only affect the directly coupled class C_2, but also the indirectly coupled class C_1. We can generalize the same by considering an extreme case where a number of classes are indirectly coupled in a chained manner. Consider a circular chain of coupled classes (class C_i uses class C_{i+1} for i = 1, 2, ..., n – 1, and class C_n uses C_1). By considering both export and import coupling it can be identified that each class in this chain is directly coupled with two of the other classes. Also, it can be observed that each class in the chain indirectly uses and is also being used by every other class.

The work described by Briand et al. [1997] was based upon high-level design measures for coupling and cohesion in object-based systems. The above work was also validated with respect its capability in identifying the fault-proneness of modules. The metrics to meaure coupling described in above work included measures for direct and indirect coupling. The direct coupling measures were found to be useful predictors, but not those for indirect coupling. The work of Briand et al. [1997] has primarily been defined to derive coupling measures for the identification of fault-prone classes, they did not include the distinction between direct and indirect coupling.

5.1.4 Existing Coupling Measures and Importance of Indirect Coupling Measure

Various coupling measures have been defined by different researchers in the literature. In 1992, Chidamber and Kemerer [1991] defined numerous metrics for object-oriented software design; some of them were related to coupling. Three different types of relationships were identified by Eder et al. [1994] identified namely interaction relationships between methods, component relationships between classes, and inheritance between classes. Their work identifies different dimensions of couplings by using these three relationships according to strength of coupling.

Hitz and Montazeri [1995] presented two types of coupling. The first type is object level coupling, which is determined by the state of an object; and the other is class level coupling, determined by the

state of an objects implementation. Different coupling strengths are also proposed in their study. In 1997, Briand et al. [1997] defined coupling as interactions between classes. The coupling strength is determined by the type of the interaction, the relationship between the classes, and the direction of the interaction.

Some of these important coupling measures are discussed in following section.

Coupling between objects (CBO)

Measure CBO is defined in [Chidamber and Kemerer 1991] as follows: "CBO for a class is a count of the number of non inheritance related couples with other classes." An object of a class is coupled to another, if methods of one class use methods or attributes of the other. In [Chidamber and Kemerer 1994], a revised definition is proposed: "CBO for a class is a count of the number of other classes to which it is coupled." At another place in a footnote it says that "this includes coupling due to inheritance."

Response for class (RFC)

According to original definition in [Chidamber and Kemerer 1994]: $RFC = |RS|$ where RS is the response set for the class. The response set can be expressed as

$$RS = \{M\} \bigcup_{all\ i} \{R_i\}$$

Where $\{R_i\}$ is the set of methods called by method i, and $\{M\}$ is the set of all methods in the class. The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class.

Message passing coupling (MPC)

Measure MPC [Li and Henry 1993] is defined as the "number of send statements defined in a class." According to the authors "The number of send statements sent out from a class may indicate how dependent the implementation of the local methods is on the methods in other classes." According to this definition MPC only counts invocations of methods of other classes, not invocations of the class' own methods. Also, in this method only send statements in "local methods" are counted. According to Li and Henry [1993] : "The local methods of a class constitute the interface increment." Inherited

methods are not part of the interface increment; therefore the inherited methods having send statements are not counted.

Data abstraction coupling (DAC)

Measure DAC [Li and Henry 1993] has been defined as "the number of abstract data types (ADTs) defined in a class." In this context, an ADT is a class in the system. An ADT can also be defined in some class, if it is the type of an attribute of class that class. According to the authors "The number of variables (attributes) having an ADT type may indicate the number of data structures dependent on the definitions of other classes."

Measures for Indirect coupling

Most of the coupling measures available in literature consider direct coupling only. Indirect way of coupling is accounted in another sense. This is done by defining RFC' [Chidamber and Kemerer 1991]. RFC' is the number of methods that can possibly be invoked by sending a message to a class C. This definition also includes methods of C, methods invoked by the methods of C, the methods these in turn invoke, etc. RFC_{α} [Churcher and Shepperd 1995] is another measure which counts such nested method invocations up to a specified level α. New measures can easily be derived that account for indirect coupling from measures that do not account for it (only possible if such consideration makes a sense). A relation called direct coupling can be described on a set of elements (e.g., a relation "invokes" on the set of all methods of the system, or a relation "uses" on the set of all classes of the system). Indirect coupling can be accounted by just considering the use the transitive closure of that relation.

Table 5.2 summarizes the coupling measures. For each measure, the type of coupling it uses, factors determine the strength of coupling, whether an import or export coupling measure, whether indirect coupling is accounted for, has been indicated.

Table 5.2 Different available coupling measures

Coupling measure	Coupling type	Strength of Coupling	Import/Export coupling	Indirect coupling
CBO (Coupling Between Objects) [Chidamber and Kemerer 1994]	method Invocation, attribute reference	No. of coupled classes	both	no
CBO' [Chidamber and Kemerer 1991]				
RFC_{α} (Response for Class)	method Invocation	No. of	import	depends

Coupling measure	Coupling type	Strength of Coupling	Import/Export coupling	Indirect coupling
[Churcher and Shepperd 1995]		methods invoked		
RFC [Chidamber and Kemerer 1994]				no
RFC' [Chidamber and Kemerer 1991]				yes
MPC (Message Passing Coupling) [Li and Henry 1993]	method Invocation	No. of methods invoked	import	no
DAC (Data Abstraction Coupling) [Li and Henry 1993]	type of attribute	No. of attributes	import	no
DAC' [Li and Henry 1993]		No. of distinct types	import	no
COF (Coupling Factor) [Abreu et al. 1995]	method Invocation, attribute reference	No. of coupled classes	both	no
ICP (information flow based coupling) [Lee et al. 1995]	method Invocation	No. of invocations and parameters passed	import	no
IH-ICP (information-flow-based non-inheritance coupling) [Lee et al. 1995]				no
NIH-ICP (information-flow-based inheritance coupling) [Lee et al. 1995]				no
IFCAIC [Briand et al. 1997]	type of attribute	No. of attributes	import	no
ACAIC [Briand et al. 1997]				no
OCAIC [Briand et al. 1997]				no
FCAEC [Briand et al. 1997]			export	no
DCAEC [Briand et al. 1997]				no
OCAEC [Briand et al. 1997]				no
IIFCMIC [Briand et al. 1997]	type of parameter	No. of parameters	import	no
ACMIC [Briand et al. 1997]				no
OCMIC [Briand et al. 1997]				no
FCMEC [Briand et al. 1997]			export	no
IC_OC [Arisholm 2002]	method Invocation	No. of method invocations	import	no
IC_OD			import	no

Coupling measure	Coupling type	Strength of Coupling	Import/Export coupling	Indirect coupling
[Arisholm 2002]				
EC_OC [Arisholm 2002]			export	no
EC_OD [Arisholm 2002]			export	no
CoupT [Gui and Scott 2006]	method and attribute invocation	No. of methods and attribute invocations	export	yes
CCC [Chowdhury and Zulkernine 2010]	method and attribute invocation	No. of method invocations	both	no

It is clear from the above table that only a few measures take indirect coupling into account. It suggests more investigation to be done to completely understand the nature of indirect coupling in object oriented systems.

5.2 Direct Coupling

To describe the concept of "Direct Coupling" [Yang et al. 2005] which deals with connections that are of "direct" nature, let us first consider the ambiguity of the definition given by [Briand et al. 1999], about direct coupling.

According to Briand et al.'s definition:

> *Direct coupling describes a relation on a set of elements (e.g., a relation "invokes" on the set of all methods of the system, or a relation "uses" on the set of all classes of the system). To account for indirect coupling, we need only use the transitive closure of that relation".* [Briand et al. 1999].

The problem with this definition is that it is not precise as to what relations may be considered as coupling. To understand the issue, consider the example shown in Figure 5.1.

```
class A{
 B obB = new B();
 E obE = new E();
 void methA(){
  obB.methB();
  obE.methE();
 }
```

```
class B{
 C obC = new C();
 void methB(){
  obC.methC();
 }
}
```

```
}

class C{
 D obD;
 C(){
  obD = new D();
 }
 void methC(){
  //no uses of obD
 }
}
```

```
class E{
 void methE(){
  //...
 }
}
```

Figure 5.1 Classes directly coupled in different ways

Several classes are shown that are directly coupled in different ways. Now, consider the relation "calls method", such a relation holds between the two classes if a method from one class calls a method of the other. This is the possibility of one coupling relationship. It should be noted that this relation is transitive but not symmetric. The two classes A and B are directly coupled by this relation, similar is the case for B and C, and also for A and E; however, A is not coupled to C, and nor E coupled to B or C. It can also be seen that class D is not related to any of the other classes; other classes are also not related to D. If we give meaning to the relation "creates instance", that means one class creates a new instance of another class, then that relation may hold between a number of classes, but it is in particular between C and D. Although this relationship can not be considered to be that much different from "calls method", it is believed that such distinction is sufficient. Figure 5.2 shows these relationships.

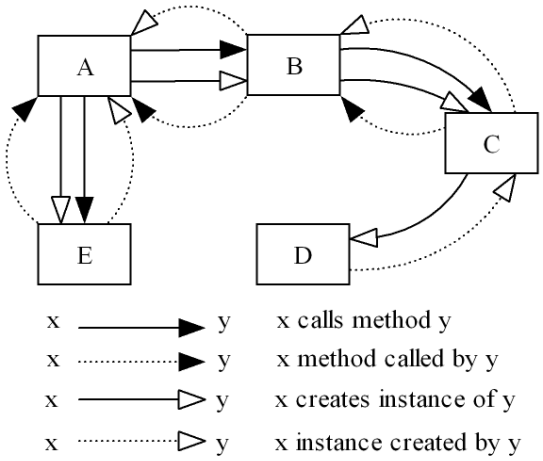


Figure 5.2 The coupling relations between the classes in Figure 5.1

If there is a new relation which can be regarded as the transitive closure of the "calls method" relation, then one may conclude that A is somehow coupled to C and it is also established that A is indirectly coupled with C since these are not coupled directly. However, even under this definition of indirect coupling still none of these classes are coupled with D. It must be noted that sometimes it is important to consider both import coupling (e.g., when some service from some module is needed by another module) and export coupling (when some service is provided by some module to another module) [Briand et al. 1999]. In the context of the above example, it can translated as whether a class contains the call to a method belonging to another class, or provides a method that is called by another class. If one consideres the export version of "calls method", call it "method called", then again it may be concluded that A, B, C are coupled, but now in the other direction, so that C is coupled to B, B to A, and E to A, but not A to C and D still is not coupled.

Consider now another relation, namely "calls method or method called", which is considered as a composition of the two relations discussed as above. Considering this relation, we get both A indirectly coupled with C and vice versa. This new relation is really just the composition of the export and import versions of the same connection, and therefore it may be argued that this kind of composition is reasonable. Further we may add "creates instance", and its export version, "instance constructed by", to the existing composition. Under this relation, A, B, C, and D all indirectly coupled together, but this may be regarded questionable since there is no justification to support the claim that A is coupled to D. But another problem arises and that is when to stop - if "creates instance" with "calls method" cannot be composed. There may be other classes also having such relationship. There remains the question to identify such relations.

The export/import aspect of the argument used above may be regarded as good criteria for allowing composition, but it also can have problems. Considering the earlier example and the "calls method or method called" which is a composite relation, it may also be concluded that, since A is coupled with E by this relation, and further A is coupled with C by using the same relation, E and C can be said to be coupled indirectly. But, in fact E and C have seemingly not related to each other, and so it seems unreasonable to consider them coupled. Therefore, it becomes difficult to justify compositing relations that are related by one being the export (import) version of the other. If we consider indirect coupling to be the transitive closure of the relation consisting of the composition of all direct

coupling relations, then it may be observed that most of the modules in a system might be coupled with each other, which does not seem very useful for understanding where designs compromise modifiability. A Simple Transitive Coupling (STC) is identified by the transitive closure of a single relation, while Composite Transitive Coupling (CTC) is identified by the transitive closure of more than one relation.

The coupling metric that takes account of the degree of coupling, functional complexity and transitive (i.e. indirect) coupling between classes in an object-oriented software system can be regarded as a directed graph [Gui and Scott 2006]. The classes comprising the system can be dnoted as the vertices in the graph. Consider a system which is comprising a set of classes $C \cong \{C_1, C_2 \ldots . C_m\}$. Let M_j represents the set of methods of the class C_j, and V_j represents the set of instance variables of class C_j. $MV_{j,i}$ represents the set of methods as well as instance variables in class C_i which are invoked by class C_j for $j \neq i$ ($MV_{j,j}$ is defined to be null). Then the edge from C_j to C_i exists if and only if $MV_{j,i}$ is not null. Therefore in its graph representation an edge of the graph reflects the direct coupling of one class to another. The graph obtained in such a manner is directed since $MV_{j,i}$ is not necessarily equal to $MV_{i,j}$. MV_j, the set of all methods and instance variables in other classes that are invoked by class C_j, can be defined as [Gui and Scott 2006]:

$$\mathrm{MV_j} = \bigcup_{1 \leq i \leq m} \mathrm{MV_{j,i}} \qquad (5.1)$$

The extent of direct coupling from class C_i to class C_j depends upon the number of methods and variables in the set $MV_{i,j}$. The class is said to be coupled strongly if this value is large.

Using the above notations we can define direct coupling from class C_i to class C_j as [Gui and Scott 2006]:

$$C_D^{i,j} = \frac{|MV_{i,j}|}{|MV_i| + |M_i| + |V_i|} \qquad (5.2)$$

In the above equation denominator represents the total number of methods and variables used by class C_i, which accounts for the total functionality of class C_i. This guarantees that the direct coupling from class C_i to class C_j, $C_D^{i,j}$ is independent of class size. In this context $directPath$ is

defined as the path between classes which are coupled as per equation (5.2). The value of $C_D^{i,j}$ in equation (5.2) will always be in the range from zero to one.

5.3 Indirect Coupling

Indirect coupling can be defined as "any coupling that is not direct coupling". This definition is appealing in its simplicity, but there are some problems associated with this definition. The first problem is, it is not an operational definition, and since we don't have an operational definition for what "coupling" is (which is basically the goal of much of the research on coupling which aims to measure some form of coupling), what we can do is use the above definition to identify a subset of indirect coupling that we can be defined in an operational manner. This is also one of the goals of the presented research.

```
class A
{
 public static void main(String[] args)
 {
  B aB = new B();
  aB.setString();
  C aC = new C();
  aC.readB(aB);
 }
}

class B
{
 String str;
 D aD;
 B()
 {
  aD = new D();
 }
 void setString()
 {
  str = aD.retVal();
 }
 String getString()
 {
  return str;
 }
}
```

```
class C
{
 void readB(B aB)
 {
  aB.getString().trim();
 }
}

class D
{
 String str;
 String retVal()
 {
  //str = "Hi";
  return str;
 }
}
```

Figure 5.3 Interpreting Indirect Coupling

Aside from STC another form of Indirect coupling is defined by [Yang and Tempero 2007b] as use-def relationships. These use-def relationships will extend from one class to some other class in a class cluster. In this particular form of indirect coupling we focus on its definition which is defined as

"*a given class C is indirectly coupled via data flow to another class D if and only if there exists a value used in class C that is defined in class D*" [Yang and Tempero 2007b]. The fundamental property of this indirect coupling is that the behavior of class C is potentially dependent on the value generated by class D. Therefore applying any changes by modifying some value defined in class D may affect the behavior of class C. For example let us consider class definition in Figure 5.3.

It is clear from Figure 5.3 that class A is directly dependent on classes B and C as it is creating their instances and calling their methods. If we try to rename classes B or C it will affect class A, as it would then require recompilation of A. Now indirect dependence is the complement of direct dependence. While indirect dependence may be thought of as just the transitive closure of direct dependences we find that this is not sufficient [Yang and Tempero 2007a]. By same definition of 'dependent', we observe that C and D are not dependent. For example, removing D from the program would not cause compilation of C to fail. However a closer inspection reveals that a different kind of dependence exists between D and C. If we execute the program (through A's main method) results in a null pointer exception thrown by C because it tries to dereference the function `aB.getString()`, which evaluates to null. This is caused because D fails to initialize the value of the field str which is used by B through `retVal()` method. Now, if we uncomment the statement `str = "Hi"` in D we can avoid the null pointer exception in C. In other words there is a change to D that affects C, which signifies dependence, and which is an indirect dependency.

This type of dependency as described above w.r.t. Figure 5.3 requires additional effort on the part of developer while performing any maintenance in the existing code. The concept of chains is introduced by [Yang and Tempero 2007b] to define the metrics for such indirect coupling. A chain can be expressed in terms of graph vocabulary. Each statement in program would correspond to a node, while each immediate data flow from a definition site to a usage site corresponds to edges. We can define "length" of chains based on the granularity level of measurement. This notion of distance can be mapped to maintenance effort, since the longer the chain of the flow of values across the system, the more work will be required to trace this flow, potentially having to switch between different methods and different classes or methods. The level of granularity can be determined by the level of boundary being considered, whether it is in terms of classes, methods or blocks. Selection of granularity of chains depends upon at what level we want to quantify the effort. In such case it is not

a straight measure and depends upon the developer. For example using distance in terms of class boundaries is a simple measure, if we want to consider the coupling interactions between the classes. But this may be inappropriate if there are various self calls within same method and this effort cannot be taken into account at this level. To account for this one has to increase granularity at method level. Therefore a trade-off is required between maintenance effort and notion of chain length based on its granularity.

5.4 Indirect Coupling Path (ICP)

In the system comprising set of classes $C \cong \{C_1, C_2 \ldots . C_m\}$ designating the vertices and direct coupling relationships as the edges of a graph as described in section 5.1, we can define Indirect Coupling Path (ICP) as a path which starts and ends in different classes, i.e. the class in which the path starts is indirectly coupled to the class in which the path ends. Furthermore, ICP is defined to be a subset of all Coupling Paths where each member path starts in a class that contains no explicit reference to the class in which the path ends, i.e. the former class is indirectly coupled to the latter. Formally we can define a Coupling Path and Indirect Coupling Path as:

$$CouplingPath = \{c | c \in Paths \wedge startClass(c) \neq endClass(c)\}$$

$$Indirect\ Coupling\ Path = \{c | c \in CouplingPath \wedge (startClass(c), endClass(c) \notin directPath)$$

Where `startClass()` and `endClass()` are the functions which return the starting class and ending class to which argument belongs.

In this research the ICP existing between various classes in a class cluster because of STC are considered. It is argued that since the maintenance effort tends to increase as the ICP length increases between any two classes, as one has to explore more number of classes. Therefore total effort will increase along with the ICP through the class. Similarly if multiple ICPs exist between any two classes, then the value of indirect coupling must consider all such existing paths instead of considering only path with highest indirect coupling value.

5.5 Relationship between Indirect Coupling and Maintenance Effort

Suppose that the two direct coupling values $C_D^{i,j}$ and $C_D^{j,k}$ for classes C_i, C_j and C_k, but the value of direct coupling $C_D^{i,k}$ is zero. Even though there is a dependency between classes C_i and C_k because C_i is depending upon C_j which in turn depends upon class C_k. Because of this indirect dependency any modification done in class C_k may affect class C_i. Therefore at the time of maintenance activities such indirect relations must be considered and the maintenance effort will depend upon the coupling path between $C_k$and C_i. This maintenance effort depends upon the fact that how strongly the individual classes are coupled together. Therefore we can define Indirect Path Coupling (IPC) between C_k and C_i as:

$$C_p^{i,k} = C_D^{i,j} + C_D^{j,k} \tag{5.3}$$

The Indirect coupling between two classes exists if there is a path from one to the other made up of edges whose direct coupling values C_D are all non-zero. The maintenance effort required depends upon the sum of all those C_D values. Thus we define Indirect Path Coupling (IPC) because of this indirect coupling between classes C_i and C_j due to a specific path *p*, as:

$$\begin{aligned} C_p^{i,k} &= \sum_{e_{s,t} \in p} C_D^{s,t} \\ &= \sum_{e_{s,t} \in p} \frac{\left|MV_{s,t}\right|}{|MV_s| + |M_s| + |V_s|} \end{aligned} \tag{5.4}$$

Here $e_{s,t}$ denotes the edge between vertices *s* and *t*.

Since direct coupling between any two classes is always be less than 1, according to equation (5.2) so indirect dependency due to longer paths will lead to increase. Longer will be the path, higher will be the indirect coupling through that path and vice versa. Here we measure Maintenance effort in terms of value of indirect coupling which is directly proportional to indirect coupling through path *p*, i.e.

Effort α Indirect Coupling

Higher the value of indirect coupling, greater maintenance effort required in tracing and modification and vice versa. Effort is associated with the sum of direct coupling measured along each path in the set of paths which exist among different software classes [Briand et al. 1999]. According to Yang [Yang et al. 2005] effort is directly proportional to length of the path. Effort is associated with the

sum of lengths measured in terms of edges in the set of edges. Therefore greater is the length of the path, more effort will be required to trace it.

Since effort is associated with the set of paths according to Yang [Yang et al. 2005], effort is directly proportional to number of paths that exists between different software classes. The length of path between any two classes having indirect coupling may or may not be same. Since, multiple path or relationship exists between source and destination, so effort required in getting the work done will be more. Similarly if there are multiple paths existing between different software classes, which may or may not have an overlapping edge in common and even path length may vary, then maintenance effort required in tracing the path and modification will cost more. Thus, effort required in this situation can be determined using the following algorithm shown in Figure 5.5. This algorithm considers that if there are multiple paths between two classes and partially they are overlapping then redundant effort for such common edges must be removed.

Below introduces functions *pathsFrom*, *pathsTo* and *pathsBetween*, all of which return a subset of ICP. *pathsFrom(C)* returns paths that start within class *C*; *pathsTo(C)* returns paths that end within class *C*; *pathsBetween(C_1, C_2)* returns paths that start within class C_1 and end within class C_2.

$$pathsFrom(C) = \{p | p \in IndirectCouplingPath \wedge C = startClass(p)\}$$

$$pathsTo(C) = \{p | p \in IndirectCouplingPath \wedge C = endClass(p)\}$$

$$pathsBetween(C_1, C_2) = pathsFrom(C_1) \cap pathsTo(C_2)$$

Figure 5.4 helps to illustrate the relationship between paths and their constructed indirect coupling graphs. This figure shows a set of classes represented by C_1, C_2, ...C_8, methods represented by *a*, *b*, *c* ...*n*, arrows represent direct coupling (edges) and paths comprising of edges. All paths identified from this program below, are those involved in indirect coupling:

The set of all ICPs= *{(a,d,g,l),(a,d,h,m),(b,e,i,n),(c,e,i,n),(d,f,j,m),(d,f,k,n)}*

- *pathsFrom(C_1)={(a,d,g,l),(a,d,h,m),(b,e,i,n)}*
- *pathsFrom(C_2)={(c,e,i,n),(d,f,j,m),(d,f,k,n)}*
- *pathsTo(C_7)={(a,d,g,l),(a,d,h,m),(d,f,j,m)}*
- *pathsTo(C_8)={(b,e,i,n),(c,e,i,n),(d,f,k,n)}*
- *pathsBetween(C_1,C_7)={(a,d,g,l),(a,d,h,m)}*

- *pathsBetween(C_1,C_8)={(b,e,i,n)}*
- *pathsBetween(C_2,C_7)={(d,f,j,m)}*
- *pathsBetween(C_2,C_8)={(c,e,i,n),(d,f,k,n)}*

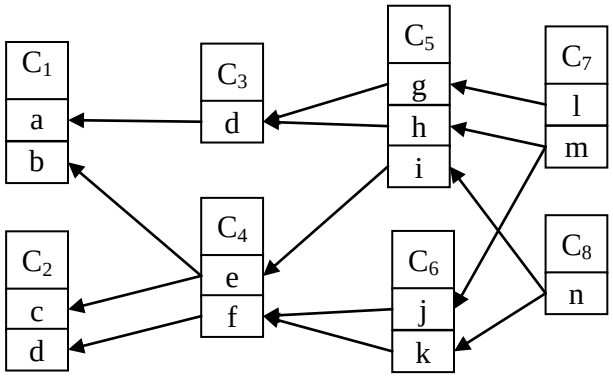


Figure 5.4 path illustration

Assume for the two classes C_i and C_j various coupling paths p_s (s > 0) exist and partially they may be overlapping. Each path in p_s may consists of a certain number of overlapped edges because of multiple paths between classes $C_i$and C_j. Each edge which corresponds to direct coupling between any two class is also associated to be a Boolean variable denoted by r_d and initialized to *false*. Here n_s is the total number of edges along path some path $p \in p_s$. If the indirect path coupling along path p is denoted by C_p then the algorithm can be written as shown in Figure 5.5.

```
for each path p ∈ p_s = pathsBetween(C_i, C_j)
   for each edge d ∈ directEdges(p)
      initialize r_d = false
                                  i = startClass(d)
                                  j = endClass(d)
                        Compute C_D^{i,j} = |MV_{i,j}| / (|MV_i| + |M_i| + |V_i|)
   endfor
endfor
for each path p ∈ pathsBetween(C_i, C_j)
   for each edge d ∈ directEdges(p)
      if r_d == false
         C_p(new) = C_p(old) + C_D^{i,j}
                                  r_d == true
      endif
   endfor
endfor
```

Figure 5.5 Algorithm for computing indirect path coupling C_p when multiple, possibly overlapping coupling paths exist between two classes.

Thus increasing the number of paths or connections between different software classes will increase indirect coupling which results in increase in the maintenance effort required in modification or extending the functionality of the class and tracing the path.

5.6 Experimental Setup

The purpose of this section is to introduce the software used for case studies and the methodology to validate the proposed metrics by data acquisition and metrics calculation. These softwares have been developed using OO programming in Java and available on www.sourceforge.net or www.apache.com. These softwares are from different functional domains having reasonable number of classes of different sizes, easy access to its source-code and availability of its different versions with the release notes made us select them as input software for our planned experiments.

5.6.1 Softwares Considered for Case Studies

This section describes the various softwares considered for case study.

EasyMock

EasyMock is the name of an open source software that we have used as an input to perform our metrics calculation. EasyMock is an open source library that provides an easy to use API to generate mock objects for given interfaces. Mock objects are created dynamically, allowing the returning of a specific result for a specific input.

EasyMock software was started in year 2003, and up till now nineteen versions of source code have been published on software website. We have selected five consecutive versions of EasyMock (version 2.0 through version 2.4) which are released in duration of three years.

Overall number of classes has been increased from 55 in version 2.0 to 63 in version 2.4 for EasyMock. Some new classes have been added and some of the old classes were dropped from the first version to last selected version. Code modification and addition are the ways through which most of the changes are performed in the internal implementation of classes. These changes, among the subject versions, are also reflected through class coupling metrics in our study.

Hibernate

Hibernate, is an Object/ Relational Mapping solution for Java environment. The term Object/Relational Mapping refers to the technique of mapping data between object model representations to a relational data model representation. Hibernate was started in year 2001, and up till now its sixty two versions of source code have been published. We have selected fifteen consecutive versions of Hibernate (version 3.0alpha through version 3.2.5) which are released in duration of three years. Overall number of classes has been increased from 592 in version 3.0alpha to 1049 in version 3.2.5. This study considers 325 classes of Hibernate having indirect coupling.

DrJava

DrJava is a simple and powerful Java development environment. It is written in Java, and runs on any Java 2 version 1.3-compatible virtual machine. (DrJava has been tested on Windows XP, Windows 2000, Windows 98, Linux, Solaris and MacOS X.) It is available for free, and it is distributed under the open source General Public License. Links to download DrJava and its source code are available from its Web page, http://drjava.sourceforge.net

Initial version of DrJava eclipse plugin was released in 2003, and up till now seven versions of source code have been published on software website. We have selected six consecutive versions of DrJava (version 0.9.0 through version 0.9.8) which are released in duration of three years. In this study 231 classes of DrJava consisting of 34566 average lines of Java source code in each version has been analyzed.

Apache Tiles

This is an HTML templating framework based on the "Composite" model. It allows for the HTML page to be broken up into multiple pagelets, called Templates, Definitions and Composing pages. At run time the pagelets are stitched together to generate the final HTML. It is available at http://tiles.apache.org/.Initial version of Apache Tiles was released in year 2007 and up till now 17 versions of source code have been published on software website. We have selected 13 consecutive versions (version 2.0.3 through version 2.2.2) which are released in duration of six years. In this

study 35 classes of Apache Tiles consisting of 2104 average lines of Java source code for each version havs been analyzed.

Apache velocity

Apache Velocity is a Java-based template engine that provides a template language to reference objects defined in Java code. It is an open source software project directed by the Apache Software Foundation and aims to ensure clean separation between the presentation tier and business tiers in a Web application (the model–view–controller design pattern). It is available at http://velocity.apache.org/ and up till now 27 stable/beta versions of source code have been published. For this case study seventeen consecutive stable/beta versions (version 1.0.1 through version 1.7) which are released in duration of four years. Apache velocity consists of approximately 118 classes implemented and 93 classes are considered in this study with an average of 15929 lines of Java source code for each version has been analyzed.

jEdit

jEdit is a text editor for programmers, available under the GNU General Public License version 2.0. It is written in Java and runs on any operating system with Java support. It uses the Swing toolkit for the GUI and can be configured as a rather powerful IDE through the use of its plugin architecture. The source code of jEdit is available at http://sourceforge.net/projects/jedit/. Ten versions (version 4.3.0 through version 5.0.0) of jEdit releases in duration of four years have been considered in this study. Approximately 378 classes of jEdit with an average of 94863 lines of java code in each version are analyzed.

jFlex

JFlex is a lexical analyser generator for Java written in Java. It is also a rewrite of the very useful tool JLex which was developed by Elliot Berk at Princeton University. As Vern Paxson states for his C/C++ tool flex: they do not share any code though. jFlex can be downloaded from http://sourceforge.net/projects/jflex/. We have selected ten consecutive versions of jFlex (version

1.3.0 through version 1.4pre5) which are released in duration of three years. This is small size software with 21 classes with 6248 lines of average java code, which are considered in this study.

jFreeChart

JFreeChart is a Java chart library that makes it easy for developers to display professional quality charts in their applications. It is distributed under the terms of the GNU Lesser General Public Licence (LGPL), which permits use in proprietary applications. jFreeChart is available at http://sourceforge.net/projects/jfreechart/. We have selected 12 consecutive versions of jFreeChart (version 1.0.1 through version 1.0.12) which are released in duration of three years. In this study 148 classes of jFreeChart have been considered with an average of 28550 lines of java code in each version of jFreeChart.

jUnit

JUnit is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks which are collectively known as xUnit that originated with SUnit. jUnit is available at http://sourceforge.net/projects/junit/. For this study seven versions of jUnit released in duration of three years have been selected. Approximately 74 clases of jUnit with 3875 average lines of java code in each version have been considered.

5.6.2 Data Acquisition

One of the most difficult and time taking part of the experiment was the data acquisition. In this part of the experiment, a tool called Indirect Coupling and Maintenance Analyzer (ICAMA) has been developed which can analyze the various classes of a release and identify the different indirect coupling paths for each class by analyzing its Java source code and computes value of IPC for each class. Following class declaration information for each class is collected:

- Member Attribute
    - Attribute Type
    - Scope of Attribute (e.g. public, private and protected)
- Member Method

    - Parameter List
    - Return Type
    - Scope of Method (e.g. public, private and protected)

Following class implementation information is also collected for each class analyzed:

- Variables used by Member Method
    - Used Member Attribute of the class
- Types instantiated in the implementation of member method
- Methods used by Member Method
    - Classes containing these methods
- Type of return variable by the member method of class
- Variables used in parameter list in the member method of class

To gather the data and analyze the code our custom tool ICAMA uses libraries from a Java 1.5 Parser under GNU Lesser General Public License (http://code.google.com/p/javaparser/). It gathers the data specified above for each class of a release, finds dependency of a class on other classes and following data is computed for each class. The coupling relationship of a class with other classes is seen as a graph with classes as nodes and the coupling between them as edges.

- Number of indirect coupling paths
- Details of each indirect coupling path
    - List of involved classes
    - Number of edges
    - Nodes (Classes) connected through each edge (coupling)
- Common edges, if any between any two indirect coupling paths

The above information is used to compute IPC for each class in a release using equation (5.4) and algorithm in Figure 5.5.

5.7 Case Studies

The proposed metrics are empirically evaluated on different releases of source code in open-source projects from industry. A total of nine open-source Java projects are analyzed. The case studies with different releases of these open source software are considered for validation of the proposed metrics.

These versions are released in duration of three or more years. The details of the different releases of these softwares considered in case studies are given in Table 5.3.

Table 5.3 Software considered for case study

EasyMock	**Versions**	v2.0	v2.1	v2.2	v2.3	v2.4			
	No. of Classes	55	55	56	60	63			
Hibernate	**Versions**	v3.0alpha	v3.0beta	v3.0	v3.0.1	v3.0.2	v3.0.3	v3.1alpha	v3.1beta
	No. of Classes	592	677	784	793	800	802	842	857
	Versions	v3.1	v3.2	v3.2.1	v3.2.2	v3.2.3	v3.2.4	v3.2.5	
	No. of Classes	909	971	1032	1034	1045	1045	1049	
DrJava	**Versions**	v0.9.0	v0.9.2	v0.9.3	v0.9.4	v0.9.5			
	No. of Classes	349	402	403	415	510			
Tiles	**Versions**	v2.0.3	v2.0.4	v2.0.5	v2.0.6	v2.0.7	v2.1.0	v2.1.1	v2.1.2
	No. of Classes	83	84	85	88	88	109	118	122
	Versions	v2.1.3	v2.1.4	v2.2.0	v2.2.1	v2.2.2	v3.0.0		
	No. of Classes	122	123	176	150	150	122		
Velocity	**Versions**	v1.0	v1.0.1	v1.1	v1.2	v1.3	v1.3.1	v1.4	v1.5
	No. of Classes	151	152	160	176	184	185	196	214
	Versions	v1.6	v1.6.1	v1.6.2	v1.6.3	v1.6.4	v1.7		
	No. of Classes	229	229	229	229	229	236		
jEdit	**Versions**	v4.3.0	v4.3.1	v4.3.2	v4.3.3	v4.4.1	v4.4.2	v4.5.0	v4.5.1
	No. of Classes	483	483	483	483	504	504	520	520
	Versions	v4.5.2	v5.0.0						
	No. of Classes	520	535						
jFlex	**Versions**	v1.3.0	v1.3.1	v1.3.2	v1.3.3	v1.3.4	v1.3.5	v1.4pre1	v1.4pre3
	No. of Classes	39	39	39	43	43	43	42	42
	Versions	v1.4pre4	v1.4pre5						
	No. of Classes	56	54						
jFreeChart	**Versions**	v1.0.1	v1.0.2	v1.0.3	v1.0.4	v1.0.5	v1.0.6	v1.0.7	v1.0.8
	No. of Classes	736	491	491	500	503	514	538	538
	Versions	v1.0.9	v1.0.10	v1.0.11	v1.0.12				
	No. of Classes	538	544	559	561				
jUnit	**Versions**	v4.5	v4.6	v4.7	v4.8	v4.8.1	v4.9	v4.10	
	No. of Classes	130	137	152	154	154	160	162	

Across all these projects it was required to categorize the changes that were made in each version into set of steps that could be used to describe how the maintenance activities performed. Steps of code changes seek to describe changes in functionality, where functionality is defined as the scope of things which the program is able to do. For example if a new conditional is added to deal with negative numbers, then the functionality of the program has increased. Conversely if that same conditional is later removed the functionality has decreased. There are some steps of code changes that do not change functionality. These steps aim to be able to completely describe the differences between two versions of a java source file. The maintenance effort has been measured by the number of lines changed per class (Change metric) in duration of three years of software development and maintenance [Koten and Gray 2006; Zhou and Leung 2007]. A line change could be an addition or a deletion. A change of the content of a line is counted as a deletion and an addition.

The case study performed over EasyMock from versions v2.0 through v2.4 which has been released in duration of three years, it is assumed that a continuous maintenance process is applied throughout the various releases. Earlier versions of EasyMock were intended to combine the precise control of mock objects with the convenience of mock objects library. The most important change was observed in v2.3 in EasyMock class that a mock object could expect the same method to be called more than once and with different arguments. Constraints over arguments and other rules were now used to dispatch invocations to expectations. Such changes increase the complexity of the software and it tends to increase the cost of software maintenance. ICAMA is used to analyze different releases of the software, it analyzes each class in different versions and computes the change metric. It may also identify the different indirect coupling paths for each class by analyzing its java source code and can compute value of IPC for each class.

Figure 5.6 (a) shows the variation of Change metric with increasing Indirect Path Coupling for different classes in EasyMock. It can be observed from this plot that most of the classes (around 36) have IPC value less than 20% of maximum value. When we compare this data with Change metric it is observed that for most classes (around 50) it is within 20% of maximum. There is only a single peak in plot of change metric that signifies that one class has undergone significant change as compared to others. Overall the observation is that the change metric does not change as change in

observed IPC for EasyMock. Similarily Figure 5.6 (b) shows the variation of Change metric with increasing Indirect Path Coupling for different classes in DrJava. Here the value of change metric corresponds to IPC value in a more coherent manner and it changes significantly with an increase in IPC. Most of the classes having high value of IPC also show high average value of change metric. Hibernate and jEdit have a large number of classes and the value of change metric varies in more significant manner with IPC in Hibernate as compared to jEdit as shown in Figure 5.6 (c) & (d) respectively. In jEdit around 100 classes are there which haven't gone under change in their releases, correspondingly their IPC value also small. For classes having IPC value more than 20% of maximum the change metric has greater variations.

Figure 5.7 (a) to (d) and Figure 5.8 show the variation of Change metric with increasing IPC value for jFlex, jFreeChart, Apache Tiles, Apache Velocity and JUnit respectively. The detailed result data for all the above discussed softwares has been tabulated in appendix A.The change metric is used in this study as measurement of maintenance effort of the object oriented software. The IPC metric is calculated for each source code base and the results consider if a correlation exists between the metric and the change identified per class.

It is observerved from these figures that there are some classes which undergo heavy maintenance as compared to other classes. Also, the number of classes which received 50% or more maintenance is within 10% of the total classes. Based on the *variation pattern* of maintenance we may categorize software in following categories:

- ***Category 1***: Shift in maintenance range is observed for a certain range of classes.
- ***Category 2***: Almost linearly varying pattern is observed for the classes.
- ***Category 3***: No specific variation pattern is observed.
- ***Category 4***: Maintenance required only after a threshold value of IPC

These observations help us to understand the relation with required maintenance and IPC metric. This relation can help to categorize the software into different categories. Once these categories are defined then for these defined set of categorized software, specific design rules can be specified to get more maintainable software.

5.7.1 Correlation

Different techniques are used to analyze the resulted data from the case studies. The data has been shown in tabular form and graphs. Pearson's Correlation Coefficient has been used to analyze the linear relationship among subject metrics.

Pearson's Correlation Coefficient is measurement of strength of linear relationship between any two variables X and Y. Pearson's Correlation Coefficient 'r' lies in the interval of -1 to 1 inclusively. i.e. $-1 \leq r \leq 1$ where r represent Pearson's Correlation Coefficient.

The direction of correlation between X and Y is indicated by negative or positive sign of coefficient. A negative sign indicated an inverse relationship. On occurrence of an inverse relationship one variable increases while the other variable decreases and vice-versa. Whenever the magnitude of coefficient becomes 1 it refers to the perfect relationship between two variables and the coefficient of magnitude 0 indicates the absence of relationship between two variables [Richard 1999].

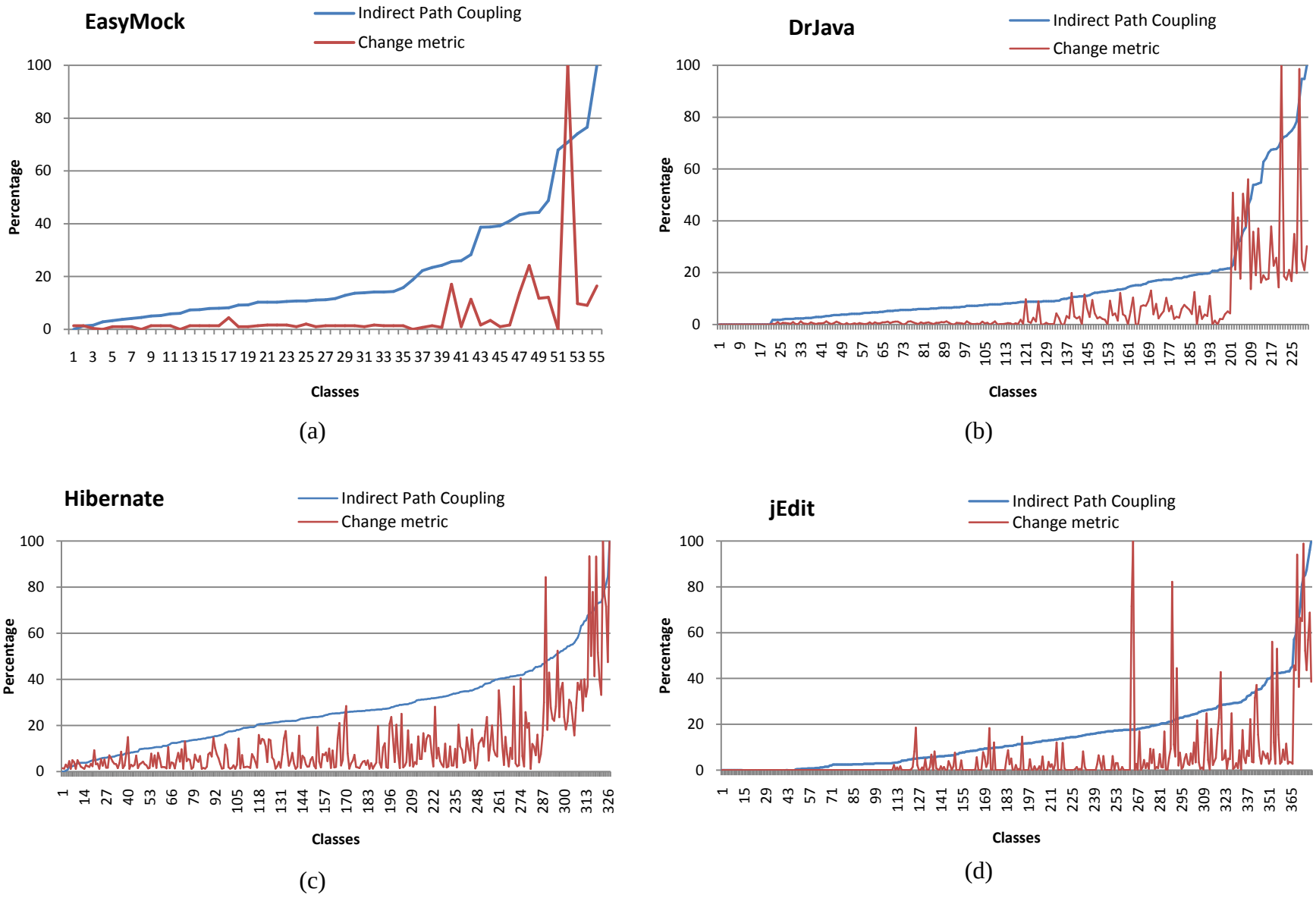
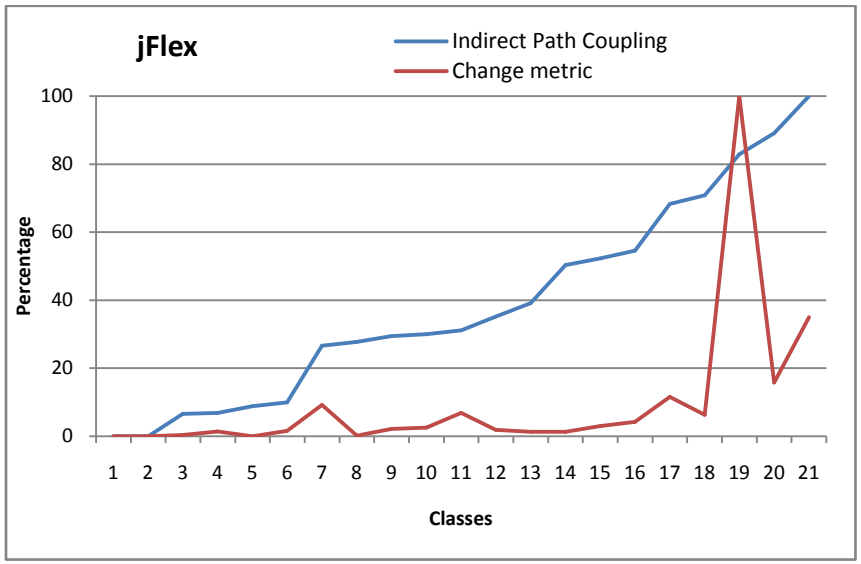
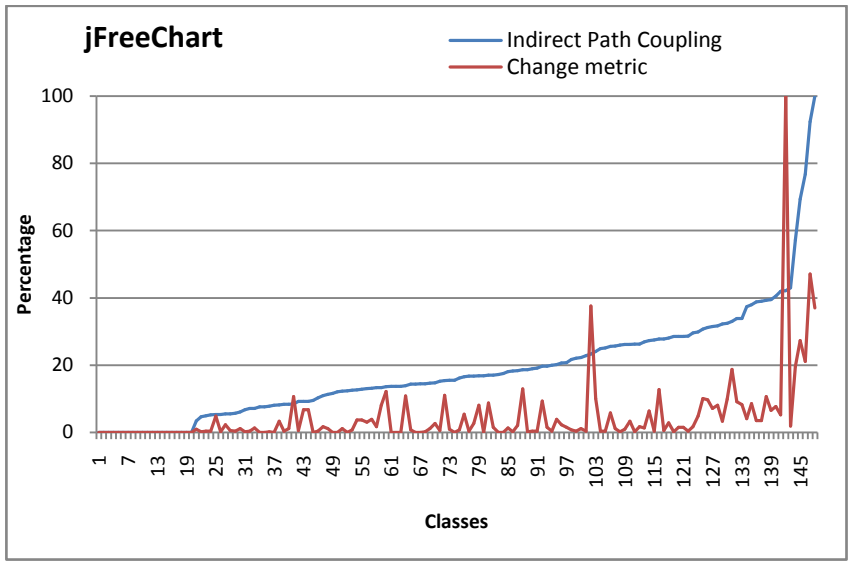


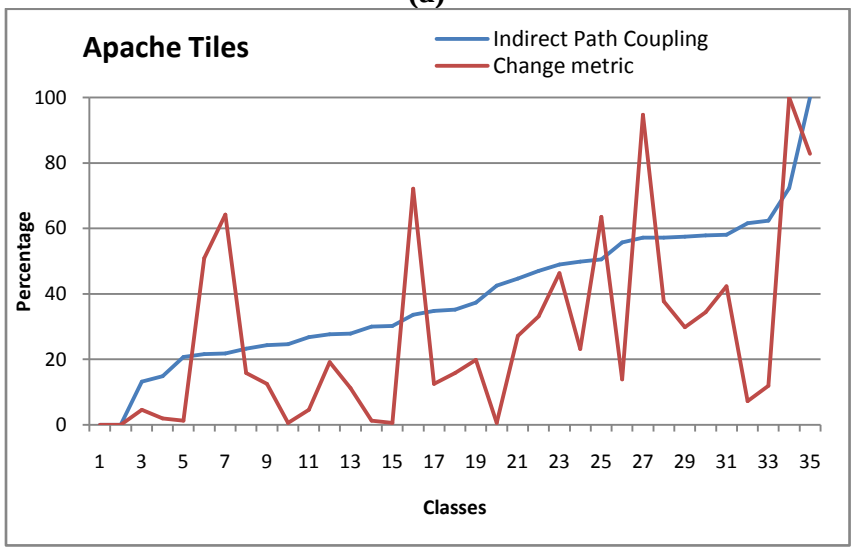
Figure 5.6 Variation of Change metric with increasing Indirect Path Coupling for different classes in (a) EasyMock (b) DrJava (c) Hibernate (d) jEdit



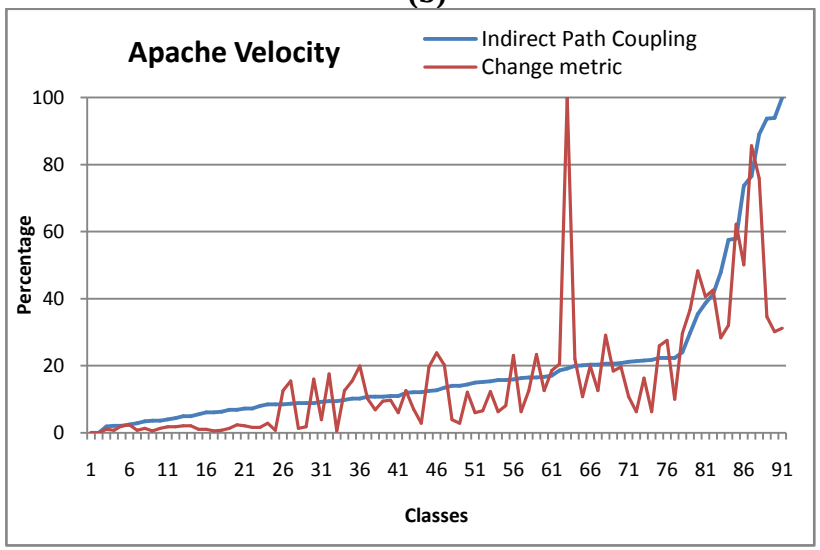
(a)



(b)



(c)



(d)

Figure 5.7 Variation of Change metric with increasing Indirect Path Coupling for different classes in (a) jFlex (b) jFreeChart (c) Apache Tiles (d) Apache Velocity

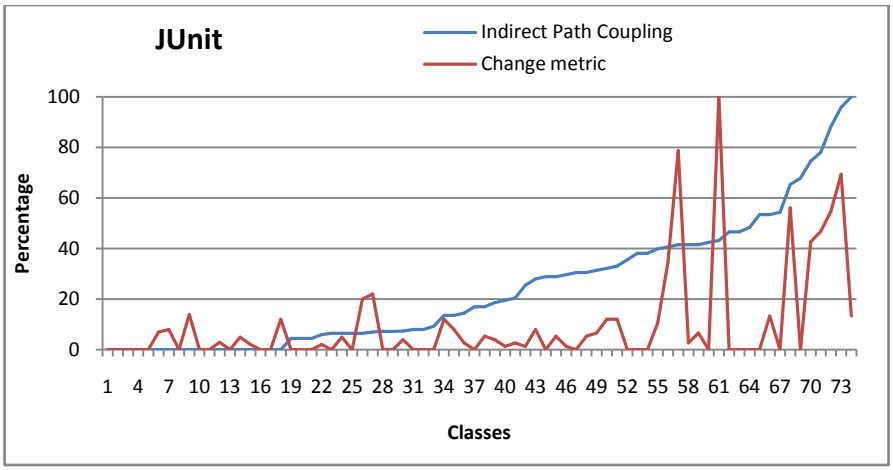


Figure 5.8 Variation of Change metric with increasing Indirect Path Coupling for different classes in JUnit

According to [Cohen 1988], assuming a large data set, the correlation of magnitude 0.5 is large, the correlation of magnitude 0.3 is moderate and the correlation of magnitude 0.1 is small. Hopkins ranks the interval of correlation, according to him, correlation magnitude $r \leq |1.\ 0|$ is trivial and it is as good as garbage, correlation magnitude 0.1 - 0.3 is minor, correlation magnitude 0.3 - 0.5 is low, correlation magnitude 0.5 - 0.7 is moderate, the correlation magnitude 0.7 - 0.9 is very good and correlation magnitude 0.9 - 1 is almost perfect [Hopkins 2003].

5.8 Data Analysis and Validation

To study the relationship between indirect path coupling and maintenance effort, the graphs have been plotted between computed indirect path coupling and the change metrics for the various versions of source code and also correlation coefficient has been calculated. The correlation coefficient for these softwares is tabulated in Table 5.4.

Table 5.4 Correlation coefficient for various softwares in study

Software	Correlation coefficient
EasyMock	0.5
DrJava	0.722
Hibernate	0.66
jEdit	0.649
jFlex	0.574

jFreeChart	0.56
Apache Tiles	0.573
Apache Velocity	0.7
JUnit	0.57

Figure 5.9, Figure 5.10 and Figure 5.11 demonstrate the correlation of proposed metric Indirect Path Coupling and maintenance effort (measured through change metric) for different software considered in this study. These graphs are plotted using the data from Appendix A. Indirect path coupling value computed for different classes is taken on x-axis while the change metric computed for corresponding class is taken on y-axis of the plot.

Using the Spearman's correlation coefficient we get correlation coefficient greater than 0.5 for all the softwares in study. For DrJava and Apache Velocity the correlation is found to be very good while for others it is moderate. With the first case study with EasyMock which has 55 classes in initial version taken and increased to 63 in last version, aroung 35 classes are having IPC value less than 5 and they also have undergone fewer changes in further releases. Only a few classes are there which have high value of IPC, but they also have undergone considerable amount of changes thereby causing more maintenance effort in future releases. For DrJava a very good correlation is found between IPC and maintenance effort. Other case study performed on open source software Hibernate over various releases from version 3.0 alpha through version 3.2.5 in duration of three years. The difference of this study with earlier case study with EasyMock and DrJava is that Hibernate is a large project as compared to them. The number of classes analyzed in Hibernate is 327 which are much larger as compared to 55 in EasyMock and 230 in DrJava. Here the correlation coefficient is 0.66 for this study. Similarily the graph is drawn for other softwares and the results of this study for computing change metric and indirect path coupling are given in Appendix A. Based on results from correlation, we argue that there is a strong correlation between IPC value and maintenance effort of the software.

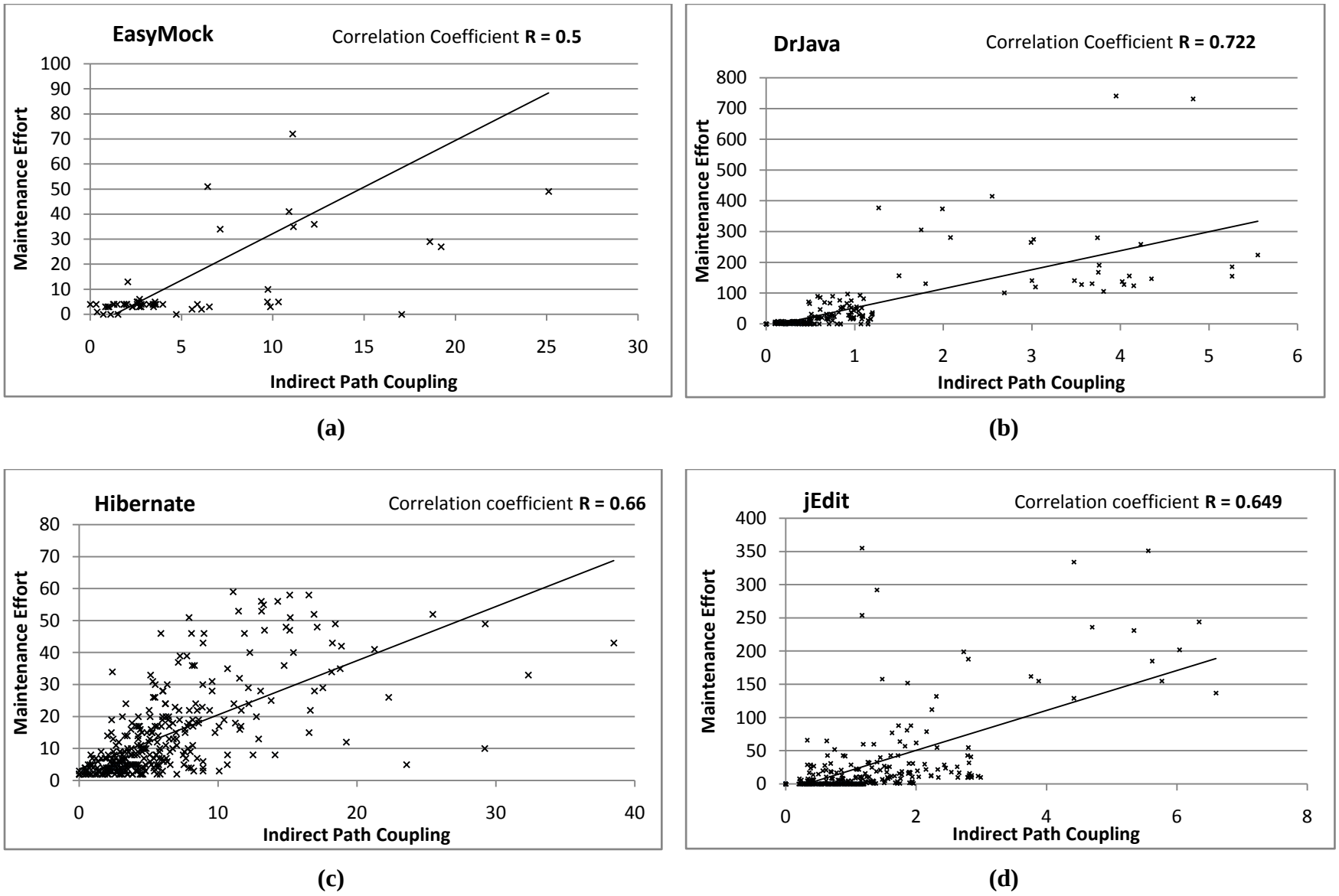


Figure 5.9 Correlation of indirect path coupling against Maintenance effort for (a) EasyMock (b) DrJava (c) Hibernate (d) jEdit

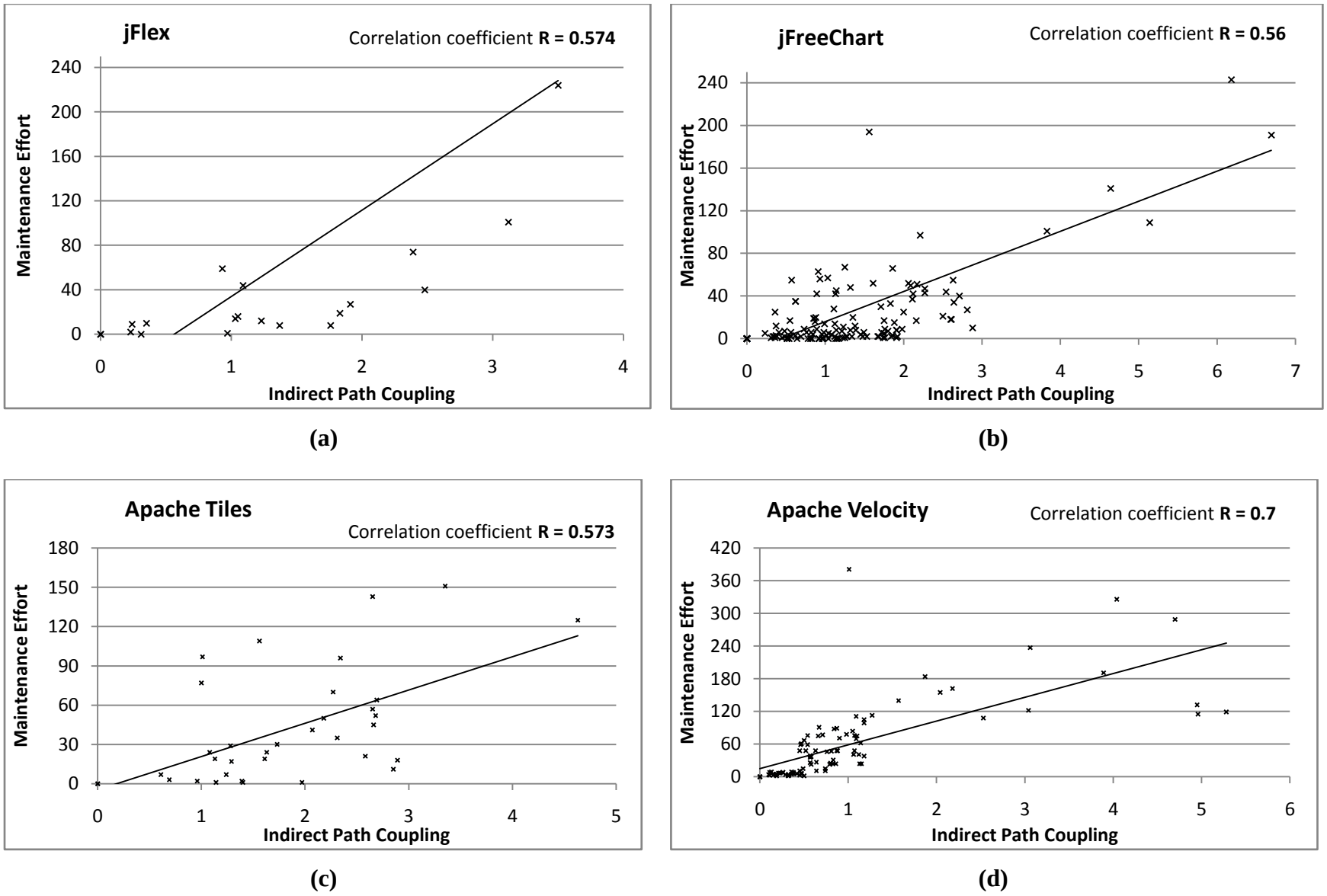


Figure 5.10 Correlation of indirect path coupling against Maintenance effort for (a) jFlex (b) jFreeChart (c) Apache Tiles (d) Apache Velocity

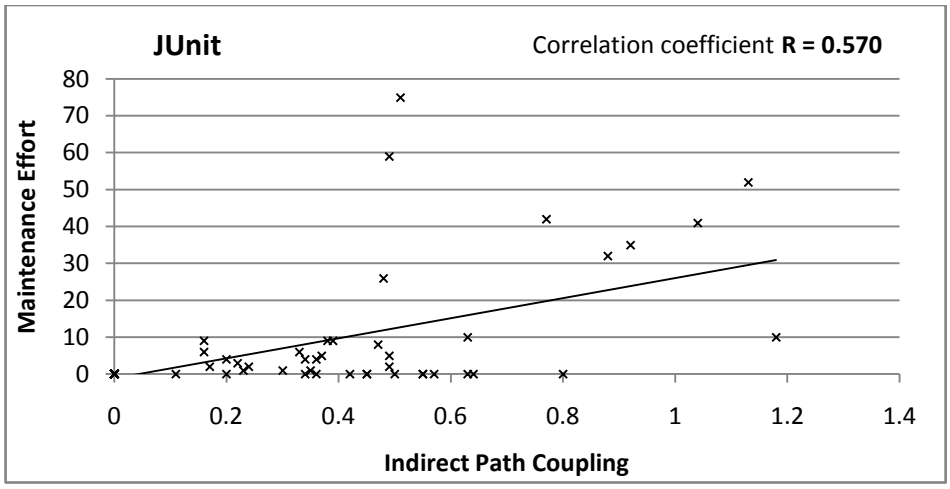


Figure 5.11 Correlation of indirect path coupling against Maintenance effort for JUnit

5.9 Threats to Validity

To ensure validity of this study, the design and methods recommended by Yin [2002] have been followed, and in this section we discuss the possible threats to validity.

Construct validity which refers to the degree to which inferences can be made from the operationalizations in the study to the theoretical constructs on which those operationalizations were based. It ensures correct operation for the concepts in the study. It is assured that the only varying factor is software system itself, there is no other factor which may influence the outcome of this study.

Internal validity reflects the extent to which a causal conclusion based on a study is warranted. Here we can say that there is no threat to internal validity as an exploratory study has been performed instead of explaining causal relationships. Interference because of any person which can lead to difference in result is not possible, as the results are not dependent upon the date or person in study.

External validity ensures the generalizability of the finding in this research beyond to the immediate case study. The most obvious open source software projects have been considered in this study. These are expected to represent a good fraction of software which is used in practice. Also there are a

large number of software available in commercial domain. In this study the selected software are from different categories and varying sizes selected randomly from open source software repository.

Reliability ensures that the operations involved in the study, such as the data collection procedure can be repeated to give the same results. It is very important as at a later point of time; if some investigator decides to repeat the experiments same results and conclusions should be obtained following the same procedure. A straightforward and simple approach has been followed, with documenting all the important deisions and intermediate results as well as the procedures for the analysis. Link to additional resources that contain additional details are given at appropriate places.

5.10 Summary

In this chapter STC form of indirect coupling has been considered which is significantly correlated with maintenance effort for systems having little interactions. Different versions of nine open source software are considered and the proposed metric is measured using various tools and it is correlated with maintenance effort which has been derived from the analysis of source code of different versions of the softwares taken in case study. The two results are correlated on scatter plot and the computed correlation coefficient validates the proposed metric with the maintenance effort.

The next chapter proposes a technique based on genetic algorithms for generating test cases in object oriented software class unit.

CHAPTER 6 *Improvements in Automated Test Data Generation Techniques*

Extensive tests can only be achieved through a test automation process [Sthamer et al. 2002]. The benefits achieved through test automation include lowering the cost of tests and consequently, the cost of whole process of software development. Static analysis techniques analyze the software being tested without executing the program code, either manual or automatic. Symbolic Execution techniques are the most widely known example of static analysis to generate test data. Several studies have been performed using this technique for automation in generating test data [Carlos et al. 2008; Howden 1982] but this technique is expensive and cannot be applied properlytoprogramshaving complex structures. This thesis proposes two approaches to automate the test data generation process. The first approach proposes a test model which is based on finite state machine specification.In this work the class specification and the test model is analyzed to select a set of test data for each method of the class, and finally the test cases can be generated using other testing techniques like finite-state testing or data-flow testing. The second approach uses Genetic Algorithms (GAs) which have been successfully applied in the area of software testing. Since, previous approaches in the area of object-oriented testing are limited in terms of test case feasibility due to call dependences and runtime exceptions. In this research, an approach to automatically generate test cases for object-oriented software is presented which relies on a tree-based representation of method call sequences. This research proposes a strategy for evaluating both feasible and infeasible test cases. With the presented approach improvements have been done to the fitness function leading to the improvement of evolutionary search by achieving higher coverage and evolving more number of infeasible test cases into feasible ones.

The next section presents a test model which is based on finite state machine specification for generation of test cases at class level.

6.1 Class Level Test Case Generation

A software testing model summarizes how one should think about test development. It tells about how to plan the testing effort, what purpose tests serve, when they're created, and what sources of

information should be used to create them. Here we have extracted our test model from the formal specification. In this strategy class specification is used to obtain class state space, which is partitioned into substates. A set of states and a set of transitions among the states compose a test model. Each state is obtained through the state space partition of the class. The value of an object from source state to target state can be changed by some transition which basically consists of some method. The input space of each method, which is the sets of values for the input parameters of the method, is partitioned. The input space partition values are used with test model to obtain the test data. Finally this test data can be used for the generation of test cases. The process of generating test cases at the class level is illustrated schematically in Figure 6.1.

6.1.1 Class Specification

The problem of precisely specifiying software modules is discussed in [Parnas 1972]. An approach for generating the test cases using state space model is given in [Tse and Xu 1996]. The limitation of their work is that if the number of states become large and for each state there exists a large input space then it becomes a deciding factor in selection of test case generation method [Gupta and Saini 2008]. The Larch Interface Language [Guttag et al. 1993] may be thought of as an approach to formal specification of program modules. This approach is an extension of Hoare's ideas for program specification [Hoare 1972]. Its distinguishing feature is that it uses two “tiers” (or layers). A class specification consists of two layers, that we will call a functional tier and a conditional tier. The abstract values of objects are defined by an algebraic specification and are called as the functional tier. An algebraic specification generally consists of a number of modules. Each module is used to specify a collection of related types. The properties of the operations (or functions) related to a particular type is specified by a set of equational axioms. A base type of a class is basically the type for the abstract values of objects.

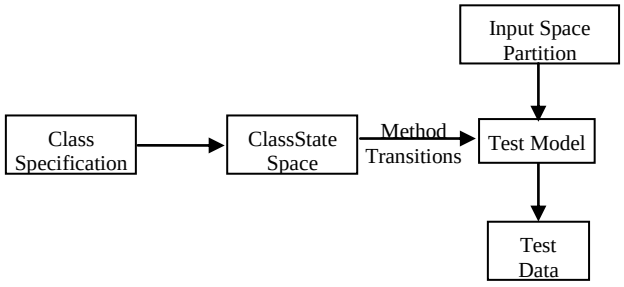


Figure 6.1 Process of generating test data

```
MinimumBalance = 1000

class Account has Balance
     operation GetBalance() returns Balance
            return Balance;
     endoperation

     operation Deposit(Amount X)
            Balance = X + GetBalance();
     endoperation

     operation Charge()
            if 500 <= GetBalance() < 1000 then
                   Balance = GetBalance() - 10;
            endif
            if 20 <= GetBalance() < 500 then
                   Balance = GetBalance() - 20;
            endif
            if 0 <= GetBalance() < 20 then
                   Balance = 0;
            endif
     endoperation

     operation Withdraw(Amount Y)
            Balance = GetBalance() - Y;
     endoperation
endclass
```

Figure 6.2 Functional tier of the class Account

Figure 6.2 shows the functional tier of the class Account in pseudo-code form, which specifies the set of abstract values for objects of this class. Class Account has various operations (methods). The operation `GetBalance` returns the current balance of any Account object. The operation `Deposit` takes some amount X and adds it to previous value of `Balance`. Every `Account` object is supposed to maintain a minimum balance of amount 1000, otherwise a charge of amount 10 is deducted if balance is between 500 and 1000. Amount of 20 is deducted if balance found less than 500. The `Charge` is applied on a monthly basis. The next operation `Withdraw` receives some amount Y. If the account has sufficient balance then this amount Y is deducted from current value of balance.

In the conditional tier, the class name, invariant, pre and post conditions of methods of each class are specified. A Class is a template for describing the attributes and behavior of its objects. At any certain moment the attribute values of an object assigned to it is called state of that object. An object can change its state by calling mutable operations. A state invariant is a condition, or constraint which is true of all possible states. Each method has its own syntax and pre- and post-conditions.

```
class Account
  invariant {Balance >= 0}
  constructor Account()
  ensures {Balance == 0}
  method Deposit (Amount X)
  requires {X > 0}
  ensures {Balance(post-operation) == Balance(pre-operation) + X}
  method Withdraw (Amount Y)
  requires {Balance(pre-operation) >= Y}
  ensures {Balance(post-operation) == Balance(pre-operation) - Y}
  method GetBalance() returns Balance
  ensures {Balance(post-operation) == Balance(pre-operation)}
endclass
```

Figure 6.3 The conditional tier of the class Account

Figure 6.3 shows the conditional tier of the class Account. The `invariant` clause specifies an invariant property that must be true of all values of the type [Leino 2008]. The invariant condition for the Account class is defined by `invariant` clause. This invariant says here that `Balance` must be non-negative in any state of `Account` object. The `method` clause declares syntax for each method. The `requires` clause is used to state a predicate that follows from the precondition. The `ensures` clause follows the post-conditions of the method. The names pre-operation and post-operation denote the value of the respective attribute before and after calling the method respectively.

6.1.2 Class State Space Partition

Partition analysis, that assists in program testing by incorporating information from both a formal specification and an implementation of procedures [Richardson and Clarke 1981]. We will improve this strategy so that it can be applied in class-level testing with formal specifications. Depending upon the values of attributes a class object may acquire various states. All such states form state space [VanderBrug and Minker 1975] of a class. In our approach, the state space of a class will be partitioned into sub states. For each sub state the input space of each method will be partitioned into sub-domains. Partition testing or sub domain testing comprises a broad class of software testing methods that call for dividing a program's input domain into sub domains and then selecting a small number of tests (usually one) from each of them [Weyuker and Jeng 1991].

Here we first consider the partition analysis of state space. Depending upon the valid values of attributes of a class it will have a certain set of states which is called its state space. Since the state

space of a class may be very large, it may become difficult to test all the states. We can subdivide it into some finite number of substates based on the class specifications. For the purpose of testing, these substates will have similar behavior. Using the state-space partition, a test model for the class can be constructed, which can further be used to generate the test cases.

Let us consider the class Account. The state space of this class is partitioned into following substates: the balance is less than 1000 and the balance is greater than 1000.

```
self.Balance < 1000, self.Balance > 1000
```

Taking into account the clauses in Figure 6.3, the state space is further partitioned into five substates. In Figure 6.4, five substates are shown.

```
self.Balance < 0 (invalid), self.Balance = 0, 0 < self.Balance < 500,
self.Balance = 500, 500 < self.Balance < 1000, self.Balance = 1000, self.Balance
> 1000.
```

Figure 6.4 The state-space partition of Account class

No further partitioning is necessary in this simple example. It is assumed that an object behaves uniformly in each substate. The test model in this example class is a finite-state machine, which describes the state-dependent behaviors of individual objects of the class. The test model has a set of states and a set of transitions along with the states. These states are obtained through the state-space partition of the class. Every transition between the states consists of a method, which changes the value of an object from the source state to the target state of the transition, and a guard predicate derived from the pre-condition of the method. There are two special states, namely initial state and final states. The initial state represents the state before the object has been created and the final state represents the state after an object is destroyed. A class Account has the test model as shown in Figure 6.5. There are seven states:

```
S0 = {unborn}, S1 = {b = 0}, S2 = {0 < b < 500}, S3 = {b = 500}, S4 =
{500 < b < 1000}, S5 = {b = 1000}, S6 = {b > 1000}
```

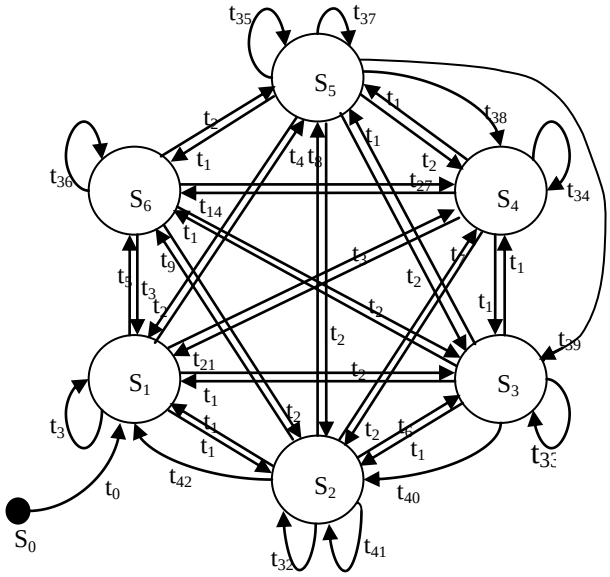


Figure 6.5 The Test Model of the class Account

Where S0 is initial state and b denotes the attribute Balance. The transitions are shown below:

t0=Account(),t1=Deposit(X) {0<X<500}, t2=Deposit(X) {X=500},t3=Deposit(X) {500<X<1000},t4=Deposit(X) {X=1000},t5=Deposit(X) {X>1000},t6=Deposit(X) {b+X=500}, t7=Deposit(X) {500<b+X<1000}, t8=Deposit(X) {b+X=1000}, t9=Deposit(X) {b+X>1000}, t10=Deposit(X) {500<X<1000}, t11=Deposit(X) {X=500}, t12=Deposit(X) {X>500}, t13=Deposit(X) {b+X=1000}, t14=Deposit(X) {b+X>1000}, t15=Deposit(X) {X>0}, t16=Withdraw(Y) {b-Y=0}, t17=Withdraw(Y) {0<Y<500}, t18=Withdraw(Y) {Y=500}, t19=Withdraw(Y) {b-Y=500}, t20=Withdraw(Y) {0<b-Y<500}, t21=Withdraw(Y) {b-Y=0}, t22=Withdraw(Y) {0<Y<500}, t23=Withdraw(Y) {Y=500}, t24=Withdraw(Y) {500<Y<1000}, t25=Withdraw(Y) {Y=1000}, t26=Withdraw(Y) {b-Y=1000}, t27=Withdraw(Y) {500<b-Y<1000}, t28=Withdraw(Y) {b-Y=500}, t29=Withdraw(Y) {0<b-Y<500}, t30=Withdraw(Y) {b-Y=0}, t31=GetBalance(), t32=GetBalance(), t33=GetBalance(), t34=GetBalance(), t35=GetBalance(), t36=GetBalance(), t37=Charge() {510<b<1000}, t38=Charge() {b=510}, t39=Charge() {500<b<510}, t40=Charge() {b=500}, t41=Charge() {20<b<500}, t42=Charge() {b<=20}.

During partition analysis, we need to distinguishbetween four kinds of methods: mutators (such as `Deposit (Amount X)` and `Withdraw (Amount Y)`), observers (such as GetBalance()), constructors (such as `Account ()`), and destructors. We need to unfold the complex denotations by introducing observable contexts [Bernot et al. 1991]. Our test model can be turned into a complete

finite-state machine by adding error states, error transitions, undefined states, and undefined transitions.

6.1.3 Partition of Input-Space

Input space of each method is also required to partition when we partition the state space of the class. The input space is the sets of values for the input parameters of the method. A valid input space for a method is the subset of the input space satisfying the pre-condition of the corresponding method. The input space of a method can be partitioned at least into two sub-domains, whether valid or invalid values. Test data can then be drawn from each sub-domain [Chen et al. 1998].

While considering the partition the input spaces of methods it becomes important to consider the type of each input variable. In above discussed example of class account, the input space of `Deposit (Amount X)` or `Withdraw (Amount Y)` the type of input is amount. The pre-condition of `Deposit (Amount X)` will give its valid input space which is applied for the input values of amount. Similarly for the method `Withdraw (Amount Y)`, the pre-condition of this method will give its valid input space. The input-space partition of each method is also related with the state-space partition of the class since methods are used to manipulate the states of objects. When we do the input-space partition of a method the state or some attributes of the class may be considered as implied parameters of the method. Based on the functional tier, the conditional tier, and the test model in figures 6.2, 6.3, and 6.5, we can partition the input space of each method as shown in figure 6.6.

Input-space partition of Deposit (Amount X)

In State S1:

p1 = {0<X<500}, p2 = {X=500}, p3 = {500<X<1000}, p4 = {X=1000}, p5 = {X>1000}

In State S2:

p6 = {0<X<500 and b+X<500}, p7 = {0<X<500 and b+X=500}, p8 = {0<X<500 and 500<b+X<1000}, p9 = {X=500 and 500<b+X<1000}, p10 = {500<X<1000 and 500<b+X<1000}, p11 = {500<X<1000 and b+X=1000}, p12 = {500<X<1000 and 1000<b+X<1500}, p13 = {X=1000}, p14 = {X>1000}

In State S3:

p15 = {0<X<500}, p16 = {X=500}, p17 = {X>500}

In State S4:

p18 = {0<X<500 and 500<X+b<1000}, p19 = {0<X<500 and X+b=1000}, p20 = {0<X<500 and X+b>1000}, p21 = {X=500}, p22 = {X>500}

In State S5:

p23 = {X>0}

In State S6:

p24 = {X>0}

Input-space partition of Withdraw (Amount Y)

In State S2:

p25 = {0<Y<500}

In State S3:

p26 = {0<Y<500}, p27 = {Y=500}

In State S4:

p28 = {0<Y<500}, p29 = {Y=500}, p30 = {500<Y<1000}

In State S5:

p31 = {0<Y<500}, p32 = {Y=500}, p33 = {500<Y<1000}, p34 = {Y=1000}

In State S6:

p35 = {0<Y<500}, p36 = {Y=500}, p37 = {500<Y<1000}, p38 = {Y=1000}, p39 = {Y>1000}

Figure 6.6 Partition of Input-space for each method of Account class

The test data for each input parameter of a method can be selected from every sub-domain using existing testing techniques. It is important here to make a little assumption. In the simplest manner we can assume that the method under test behaves uniformly in each sub-domain. Based on the above assumption, we need to select one value randomly which will work as representative from each sub-domain, and for each method we can obtain a set of test data.

6.1.4 Generation of Test Cases

The test cases generated in this way have various method sequence invocations utilizing the different sets of test data. The various methods of class interact to change the state of the class object. Thus the generated test cases are actually used to test the various scenarios which are dependent on the change of state which in turn is depending upon the interaction of methods. Here let us first use finite-state

testing techniques to generate test cases [Offutt et al. 2008]. A test case is generated by traversing the test model from the initial state. Method sequences are derived from the traversed transitions. A set of test cases are required to be generated so that it can cover the test model in the form of state coverage, transition coverage, and path coverage. For example, the test case

Account(), Deposit(600), Deposit(400), Withdraw(700), Deposit(1000)

covers the six states in the following sequence: S0, S1, S4, S5, S2, S6

We may also use data-flow testing techniques to generate test cases. Here the test cases will be generated according to def-use criteria [Weyuker 1982] of attributes in the test model. Each attribute used in the method is classified as being defined or used. It is called as defined at a transition if the value of the attribute is changed by the method of the transition and is said to be used if the method of the transition only refers to the value of the attribute. A set of test cases is generated according to certain data-flow testing criteria, which covers such def-use paths with regard to each attribute. Various testing techniques based on finite-state machines can be found in literature [Cheng and Krishnakumar 1983; Friedman et al. 2002; Naik 1997].

6.1.5 Analysis

In this section we discussed a technique which can be used to generate the test cases at class level testing for object oriented programs. The testing technique based on class specification is used in this method. This technique provides the test model which can be integrated to other existing techniques to generate the test cases. The generation of test cases, its execution and test result analysis can be done in a systematic and in an effective way using this test model. Since the test model is generated based on the behavior of the class as specified by the class specification, therefore the actual intended behavior is also represented by the test model.

The proposed technique can be used for:

- **a.** Implementing an object oriented test tool to generate test data.
- **b.** Research can be done for other testing techniques at cluster-level and system-level by extending the current work.
- **c.** Evaluation can be done on some real life application problems so that the achieved results could be used to compare with existing methods.

The next section applies genetic programming technique to generate the test cases for object oriented software.

6.2 Genetic Programming Technique for Test Case Generation

The Genetic Algorithms based on principle of natural evolution are generalizations of the approach which can be actually implemented by computers are known as Genetic Programming (GP) approach [Koza 1992]. In our genetic programming approach we represent the test cases as trees. A test case in object oriented program is a sequence of method calls. A test case in object oriented software contains the numeric test data in form of parameters as well as the sequence of constructor and method calls is also necessary. The reasons for this are following:

1. Multiple objects may be involved in a single test case. Therefore the additional objects may be required as parameters or whose methods may be required for Class Under Test (CUT) for execution of its test case. Further it may also be the case that creation of these objects may need to create more additional objects. Test cluster classes are the set of all these classes from which different class instances are required.
2. In order to process any particular test scenario in desired way the objects must be taken to some specific state (e.g. using code coverage criteria certain objects must be in a particular state). Therefore the participating objects are required to be put in those special states to process the test case. Then method calls will be required to be issued for the test cluster objects.

Therefore, a test case for any object oriented software is required to consist of the definition of testing prerequisites, a test program consisting parameters, their types and their values, method calls as well as the test oracle. This test oracle is used for validation of test results. Each test case can be considered as a sequence of statements $S=\{s_1, s_2, \ldots s_n\}$. A statement consists of the following essential components:

- target object
- method
- parameters
- receiver

The above specified information will be needed to encode a genotype individual or chromosome. A method in a statement can be a class method or it may be a constructor. The component parameters

will hold the type of the variable. The type of variable in parameter may be primitive or also it may be user defined. The corresponding values of associated types are also required. The method which holds the parameters with their types and values are called on the target object in a statement. The value returned by the method of the target object is hold by the receiver component. Typically any statement s_i can only be executed if an appropriate target object and all the required parameter objects for its method have been created in advance, i.e. during the calls of s_1 to s_{i-1}. Obviously there exists a call dependency among the methods which must be taken into account when writing statements for test cases. Therefore it may be concluded that any arbitrary sequence of statements is not feasible, but only those which respect the call dependencies.

```
Algorithm
Input: class under test
Output: test case suite
For each class under test do
    For each method under test do
        Generate method call sequence
        Encode to tree representation
        Create CFG of method
        Instrument method as per CFG
        Create Initial Random Population
        Repeat
            For each individual in population do
                Decode to method call sequence
                Synthesize Test program
                Compile and execute test program
                Compute fitness
                Compute coverage
            Select individuals to reproduce
            Apply Crossover, Mutation
            Select next generation
        Until stopping criteria is met
```

Figure 6.7 Algorithm for class testing using proposed approach based on GA

An algorithm for test case generation based on GA and has been used in this research is given in Figure 6.7. A flowchart is also shown in Figure 6.8 explaining this test data generation process.

6.3 Encoding and Decoding of Chromosome

The components of each test case statement are encoded using genes while chromosomes are constructed. This encoding assigns some number to each corresponding gene. At the time of decoding, the integer values of the genes identify the components such as method to be called and the target object to be used for the invocation. Usually the methods in a test cluster have parameter lists

with different lengths; this requires that multiple genes to be assigned to represent the parameters for a method. The data type for a parameter gene depends on the parameter to which it is assigned.

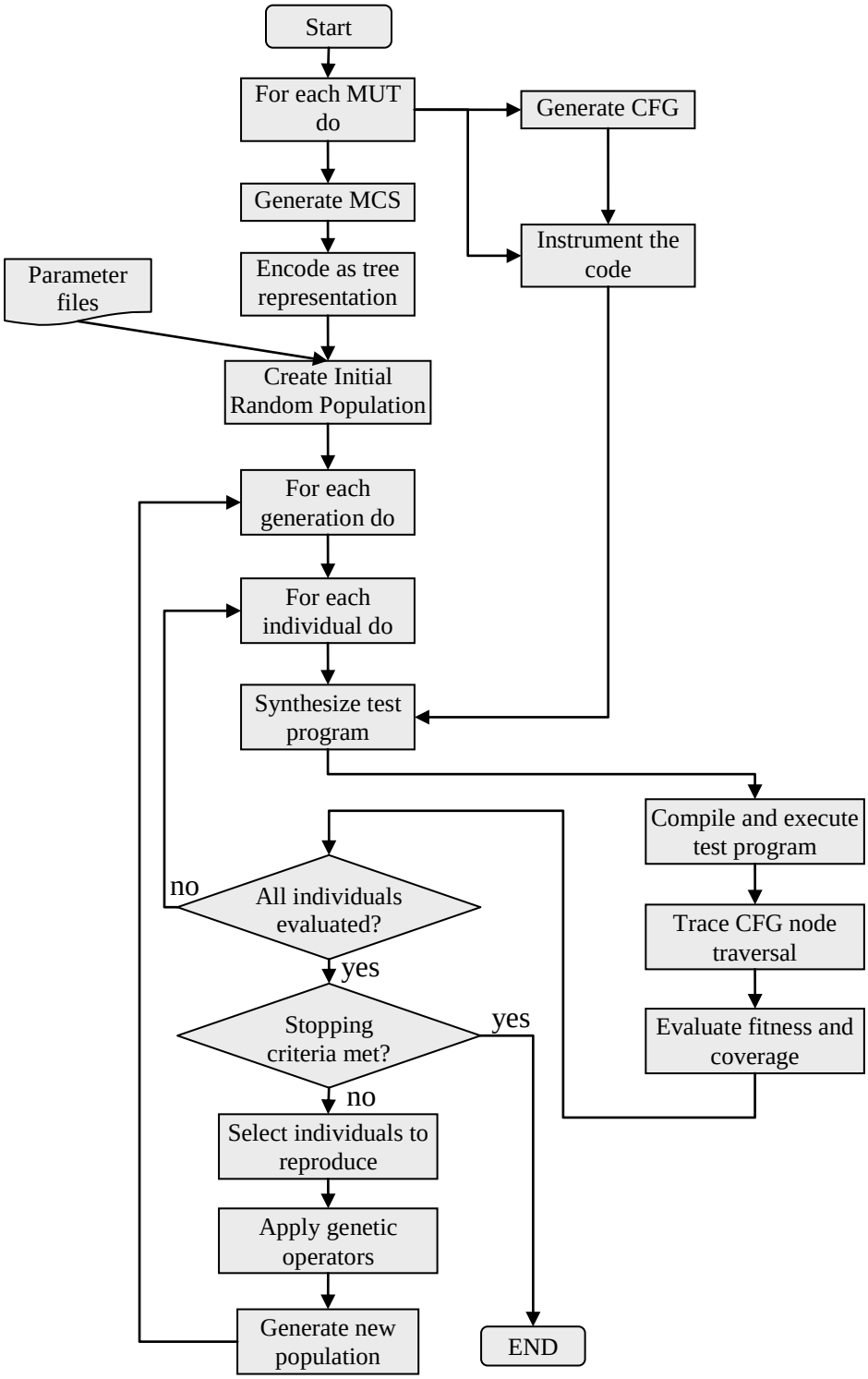


Figure 6.8 Flowchart for test data generation using proposed approach based on GA

If the assignment is not clear at the time of encoding, that data type must be used which would allow for every possible decoding. Furthermore, a statement from a test program can be represented by a gene indicating the method to be called, a gene indicating the target object for which to call this method, and a number of variables which are used as parameters for the method call. A gene indicating the receiver object is also assigned if it exists in the statement.

These all such genes will make a chromosome, which will form the various nodes of the tree representation of test case.

6.3.1 Encoding

For the genetic algorithm techniques to be applied, all the components of a test case statement are must be encoded into genes to form a chromosome. For encoding the constructor and methods they are serially numbering to form an ordered set of constructors and methods in the test cluster classes. In the chromosomes a gene is assigned to represent a constructor or a method appearing in test program. The domain *D(M)* for the genes can be defined as maximum number of test cluster methods:

$$D(M) = [1, | S_C \cup S_M |] \subset \mathbb{N} \quad (6.1)$$

Where $S_C = (c_1,c_2,....c_r)$ and $S_M = (m_1,m_2,....m_n)$is the ordered set of constructors and methods respectively of the test cluster class. $\mathbb{N}$ is the set of all natural numbers.

Similarly a gene must be assigned in the chromosome to represent a target object for which some method will be called by objects of test class cluster. In programming languages this is a kind of object reference based on natural numbers. If the number of candidate target objects for a statement s_i is known then only the domain for the genes of target object will be possible. This number is directly dependent upon the object creating methods which are called before statement s_i is called. The domain $D(T_i)$ of target genes for statement s_i can be defined as:

$$D(T_i) = [1, | O_t^i |] \subset \mathbb{N} \quad (6.2)$$

where, $t \in C$, the set of all classes in test cluster, is the required target object and i is the index of current statement, $O_t^i = (o_1, o_2, ... o_n)$ which is the ordered set of objects and are instances of class t and have been created by the statements which are called before s_i.

The encoding of receiver object can be done by assigning a gene for the receiver object used in the statement. The domain of receiver objects can be defined as the objects created by method call in statement s_i.

$$D(R_i) = [1, | O_r^i \cup B |] \quad (6.3)$$

where r $\in C$, the set of all classes in test cluster, $O_r^i = \{o_1, o_2, \ldots o_r\}$ the ordered set of objects created by method call in statement s_i. B is the ordered set of basic types available in the language. A gene is assigned in the chromosome for each required parameter. The definition of its domain depends on the data type of the parameter it represents which may be primitive or object type parameters. The domain is same as the data type ranges and precision used in language for primitive data types. For object type parameters a similar object reference mechanism can be used as for the target objects.

6.3.2 Decoding

The chromosome contains the various genes which represent the various operations as identified. The gene representing a method can be described by a function γ which maps each value of method gene to a particular method, associated with it. Suppose $m_k \in S_M$, the kth method in S_M then:

$$\gamma: G_j \rightarrow m_j$$

Where G_j is the gene which was encoded for method m_j.

Now, when decoding for target methods the values must be adjusted to the actual number of candidate target objects. If target T_i was encoded as:

$D(T_i) = R_{Ti}$

Then decoding can be done by defining a function ρ which assigns each value t of gene G_t , such that $t \in R_{Ti}$ to a target object $o \in O_t^i$:

$$\rho: R_{Ti} \rightarrow O_t^i$$

and $t \rightarrow o_t$ where $o_t \in O_t^i$

While decoding parameter genes, G_p, a function ξ can be defined which assigns value to each value of G_p which may refer to a primitive value or object reference. If it is a primitive value then no mapping will need, same value can be directly used as parameter value. If the parameter is of object type then a mapping of R_{Pi} to object reference must take place:

$$\xi: R_{Pi} \rightarrow O^{i}_{p}$$

$$p \rightarrow o_p$$

where, o_p is the pth object of the ordered set O^{i}_{p} .

6.4 Fitness Evaluation

The fitness function is constructed on the basis of the software to be tested. This fitness function itself is not of interest for the problem, but here the goal is to find test data that fit the test criteria. A well-constructed fitness function can:

- effectively increase the chance of finding the optimal solution and a better coverage of the program under test is obtained and
- search process is better guided and therefore the solution optimizes within less number of iterations.

Other work on designing fitness functions and the results of the optimization process can be found in [Jones et al. 1996], which investigates the use of various distance functions like Hamming distance, reciprocal function and their influence on optimization performance. In [Jones et al. 1996], hamming distance is considered better and the authors used genetic algorithms with a bit representation of all parameters in their approach.

Modifying the distance function of branch conditions is one of the possible mechanisms for modifying the fitness function. In our research we argue that more general alterations to the fitness function may lead to better results in Evolutionary Testing. This increases the chance to find the solution and optimization process also gives a better performance in general. Work of Baresel et al. [2002] discusses dependencies within loop iterations and introduces fitness functions with an improved behavior for optimizing test data for target nodes in loops. We use this approach to improve the methodology proposed by Carlos et al. [2008] for evaluating the quality of both feasible and infeasible test cases i.e., those test cases whose execution is completed effectively and terminate with a call to the method under test, and those tests which abort prematurely because some runtime exception has been during test case execution. In their approach, instead of simply refusing the infeasible test cases these are also considered at certain stages of the evolutionary search, therefore promoting diversity and enhancing the possibility to achieve full coverage. In their work weights of

Control Flow Graph (CFG) nodes are reevaluated by multiplying it three factors defined in their work. But their work hasn't considered the dependencies within loop iterations as described by Baresel et al. [2002]. This may cause to deteriorate the efforts done towards evolutionary testing. This may happen because guidance to appropriate test data lacks which may result in loop iteration even if it is closer to the unexplored node. In most of the cases, this situation may lead to a random search where the chance of finding a solution becomes very low if the search space is large. In the presented research we solve this problem by adding dependencies of one loop iteration to the fitness function. We can observe this information for all the iterations and fitness may be calculated from it while monitoring the execution of the test object as described by Baresel et al. [2002]. This may lead to improve the chance of finding a solution.

In the presented approach, the test case quality is related to the CFG nodes of Method Under Test (MUT) since these nodes serve as targets during evolutionary search process. Those test cases which execute an unexplored (or less explored) path or a node in CFG graph need to be favored. The basic aim of test generation process is to find a set of test cases which may achieve full code coverage of the test object in minimum number of generations. If a runtime exception occurs while executing the test case, it may be aborted prematurely. Whenever it happens, the structural behavior of MUT is impossible to be traced because the final instruction of Method Call Sequence (MCS) is unable to be executed. Therefore the test cases broadly are divided into following two categories:

- Feasible Test Cases, which are effectively executed, and are terminated by executing the final instruction of MCS.
- Infeasible Test Cases are those which terminate prematurely because a runtime exception has been thrown by some instruction of the MCS.

In general, it has been observed that longer and more intricate test cases are more prone to throw runtime exceptions [Carlos et al. 2008]. However, to elaborate state scenarios and traverse certain problem nodes complex method call sequences are oftenly needed. These complex method call sequences may generate infeasible test cases, which may need to be penalized. But blindly penalizing the infeasible test cases will discourage the definition of elaborate state scenarios. In this research, the issue of approaching the search towards the traversal of new and interesting paths and CFG nodes has been addressed by assigning weights to the CFG nodes. In this approach higher weight of a node

means that the cost of exercising it will also be higher and therefore higher cost is associated to transverse the corresponding control-flow path.

During the start of the first generation the weight of each CFG node is initialized to W_{init}, then in each successive generations the weight of each CFG node is reevaluated to accommodate the following factors:

1. The Hit Count Factor (HCF), which is computed as $\left(\frac{N_t - N_{ni}^h}{N_t}\right)$. It accounts for deteriorating the weight of recurrently hit nodes of CFG. Here N_{ni}^h parameter contains the count of how many times noden_i was exercised by the test programs of the previous generations. N_t represents the number of test cases produced in the previous generation. Here the value of HCF remains between 0 and 1. If a node is hit more number of times, HCF will be close to 0 which decreases weight of corresponding node more rapidly.
2. The Path Factor (PF) which is used to improve the weight of nodes which lead to interesting nodes and thus belong to interesting paths. We compute PF as $\left(\frac{\sum_{x \epsilon\, Suc_{ni}} \left(\frac{W_x}{W_{init}}\right)}{|Suc_{ni}|}\right)$. It computes the average value of ratio of change in weight of a node with its initial value of weight for all successor nodes of corresponding node n_i. Suc_{ni} represents set of all successor nodes of n_i in CFG graph and $|Suc_{ni}|$ represents count of all successor nodes of n_i. Therefore PF will decrease the weight of node n_i in reevaluation if the overall weight of its successor nodes has been decreased from their initial value, otherwise it will increase. That means if node n_i leads to unexplored or interesting CFG nodes then PF will increase its weight.
3. The Weight Factor (WF), we represent it as α is needed in node reevaluation because HCF will always decrease the weight of node n_i for each test case while PF may increase or decrease it. Therefore, to intensify the path search we need to ensure that the weight of node n_i should be restrained. To accomplish this we use WF whose value should be selected properly which can lead to enhance the search process taking minimum number of generations.

Therefore, considering all the above factors the weight of each CFG node is reevaluated in every generation according to following equation:

$$W_{ni} = \alpha W_{ni} \left(\frac{N_t - N_{ni}^h}{N_t}\right)\left(\frac{\sum_{x \epsilon Suc_{ni}} \left(\frac{W_x}{W_{init}}\right)}{|Suc_{ni}|}\right) \quad (6.4)$$

The fitness of feasible test cases is computed on the basis of their trace information, which includes the nodes which are hit by that test case. If H_t denotes the set of nodes which are traversed by a test case t, and thus $|H_t|$ denotes the number of nodes along this path then the fitness of this test case is evaluated as follows:

$$Fitness_F(t) = \frac{\sum_{h \in H_t} W_h}{|H_t|} \quad (6.5)$$

Using this strategy the fitness of those test cases which traverse the same path which has been traversed already deteriorates in subsequent traversals because the weight of frequently hit nodes is increased thus worsens the fitness of those test cases who execute through that path.

The fitness of infeasible test cases is computed as the ratio of weights of all the remaining possible nodes in CFG where runtime exception occurred with the weights of all those nodes which have been exercised by that test case before the runtime exception occurred. If exception occurs at node n_e and H_d^{ne} denotes the set of all nodes which are descendants of node n_e in CFG and H_t^{ne} denotes the set of all nodes which are traversed by the test case before the exception occurs.

$$Fitness_I(t) = \beta + \frac{((\sum_{h \in H_d^{ne}} W_h)/|H_d^{ne}|) * W_{init}}{(\sum_{h \in H_t^{ne}} W_h)/|H_t^{ne}|} \quad (6.6)$$

In this manner the fitness of infeasible test cases is also depending upon infeasibility factor β which is added to penalize the fitness of infeasible test cases. The infeasible test cases are selected to improve into feasible test cases at certain point of evolutionary search, which favors the diversity and complexity of MCSs. If β is large, more number of infeasible test cases may be selected for next generation which may reduce the possibility of a better feasible test case to be selected for next generation. If β is very small, then only few infeasible test cases will be selected for next generation which diminishes the overall idea of giving weights to CFG nodes for computing fitness. Therefore this value must be selected very carefully for better results.

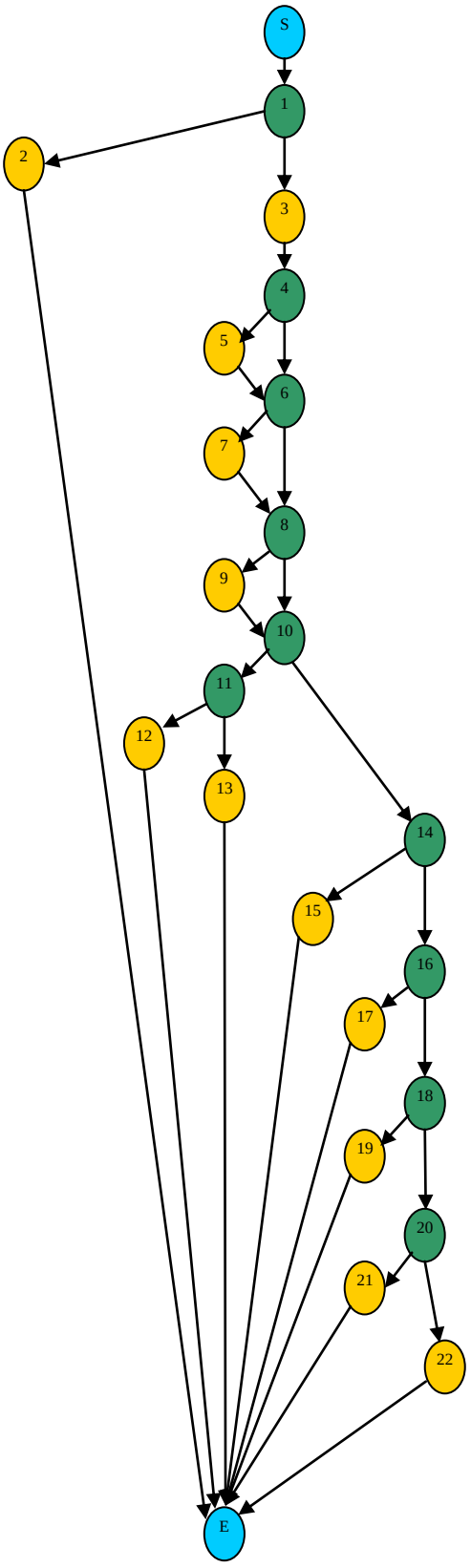
6.5 Case Studies

The approach outlined in this thesis involves the generation of test programs and data to be used as parameters of the test object and its evolution through a GA based strategy. The proposed method uses a hybrid version of ET, using characteristics of Conventional ET and Object Oriented ET, and that one goal of its construction is to improve performance of unit tests. Case study is conducted to evaluate the technique. In this study we perform experiments comparing the improvement in test cases achieved by Evolutionary Testing by varying various parameters. For the purpose of experimental study a tool EpochX v 1.4.1 (URL: http://www.epochx.org/index.php) has been used. This tool is an open source genetic programming framework. It is designed specifically for the task of analyzing evolutionary automatic programming.

6.5.1. Case Study 1

For conducting the case study the example shown in Figure 6.9 has been considered. This is a classical example used by many researchers in software testing area to test the code coverage [Ammann and Offutt 2008]. This example is a simple program for classification of triangles. The class for which test cases are to be generated is `TriangleTest` class and the method under test is `Triang()` which takes three parameters of type `Side`. If these three parameters define an invalid triangle then it has been considered as exception and program terminates.

Firstly an experiment is conducted to investigate the importance of Combination Rate (CR) and Mutation Rate (MR) on the efficiency of the genetic algorithm. Another parameter that might most obviously affect performance is the size of the population. For the experiment population size is varied among the values 100, 150, 200 and 300. To establish that the values chosen for the mutation and crossover rates were reasonable, these two parameters are jointly varied over a range of values. Initial experiments showed that mutation and crossover rates less than 0.1 or greater than 0.8 did not improve performance, so these experiments limited crossover and mutation rates to values within this range. Crossover and mutation rates were thus varied between 0.1 to 0.8 per generation. For this initial experiment the values of α and β are 1 and 0 respectively. These values ensure that they don't put their own impact while computing the fitness of feasible and infeasible test cases as their impact is considered in separate experiments.




```
S    static void Triang (Side Side1, Side Side2, Side Side3)
     {
      //  triOut and triexcept are the class variables
      //  Triang = 1 if triangle is scalene
      //  Triang = 2 if triangle is isosceles
      //  Triang = 3 if triangle is equilateral

1     if  (Side1.getSide()  <=  0  ||  Side2.getSide()  <=  0  ||
2    Side3.getSide() <= 0) {
       triexcept = true;
       return;
      }
3
4     triOut = 0;
5     if (Side1.getSide() == Side2.getSide())
6      triOut = triOut + 1;
7     if (Side1.getSide() == Side3.getSide())
8      triOut = triOut + 2;
9     if (Side2.getSide() == Side3.getSide())
10     triOut = triOut + 3;
11    if (triOut == 0) {
       if (Side1.getSide() + Side2.getSide() <= Side3.getSide()
                    ||  Side2.getSide()  +  Side3.getSide()  <=
12   Side1.getSide()
                    ||  Side1.getSide()  +  Side3.getSide()  <=
     Side2.getSide()) {
         triexcept = true;
13       return;
       }
       else
         triOut = 1;
14     return;
15    }
16
17    if (triOut > 3)
18     triOut = 3;
19    else if (triOut == 1 && Side1.getSide() + Side2.getSide() >
20   Side3.getSide())
21     triOut = 2;
      else if (triOut == 2 && Side1.getSide() + Side3.getSide() >
22   Side2.getSide())
       triOut = 2;
F     else if (triOut == 3 && Side2.getSide() + Side3.getSide() >
     Side1.getSide())
       triOut = 2;
      else
       triexcept = true;
      return;
     } // end Triang
```

Figure 6.9 CFG of method considered for case study 1 and its code

The results of these experiments are shown in Table 6.1 for different combinations of CR and MR. The coverage is computed after 200 generations for different values of population size N. From the Table 6.1 it is observed that maximum value of coverage 59.09% is achieved at CR=0.7, MR=0.2 with population size 200, other values, if exist can be rejected if corresponding population size is larger than 200.

Table 6.1 Coverage obtained for different values of population size

N	CR=0.6 MR=0.1	CR=0.6 MR=0.2	CR=0.6 MR=0.3	CR=0.7 MR=0.1	CR=0.7 MR=0.2	CR=0.7 MR=0.3	CR=0.8 MR=0.1	CR=0.8 MR=0.2	CR=0.8 MR=0.3
100	36.36	36.36	40.91	40.91	36.36	36.36	40.91	45.45	40.91
150	45.45	45.45	40.91	45.45	50.00	40.91	45.45	50.00	45.45
200	50.00	54.55	50.00	45.45	59.09	45.45	48.00	52.50	44.50
250	52.00	55.55	49.20	46.45	55.09	44.25	50.00	54.55	45.45
300	54.55	54.55	45.45	50.00	59.00	45.45	45.45	59.09	50.00

The remaining experiments are conducted by varying other parameters which are weight decrease factor α and infeasibility factor β keeping CR=0.7 and MR=0.2 which is their optimum value obtained in this case study. Table 6.2, 6.3 and 6.4 summarize the results obtained for various combinations of α and β. These experiments are conducted for a population size of 200 individuals per generation. The maximum number of generations is 200, if full coverage is achieved earlier then program terminates. The first generation which has 200 individuals is generated randomly. For crossover operation these 200 individuals are divided into two groups of 100 individuals each. Each individual from first group performs crossover operation with a single individual from second group with probability decided according to crossover rate. If crossover operation takes place between these two individuals as explained in section 6.1.4 then two offspring are also added into the current population. Therefore this operation increases the population size. The next operation which is performed is mutation and it operates as explained in section 6.1.4. For evaluation of the fitness each individual in converted back to method call sequence with instrumented MUT and is executed with associated parameters. The program traces the nodes which are covered by the corresponding test case. If the test case is feasible its fitness is computed according to equation 6.5, otherwise if infeasible then its fitness is computed according to equation 6.6. At the end of each experiment we compute improvement factor for feasible and infeasible test cases so that the most efficient combination of α and β can be identified. Improvement factor for feasible (IF_F) is computed as:

$$IF_F = \frac{N_{last}^{Feas} - N_{first}^{Feas}}{N_{first}^{Feas}} \quad (6.7)$$

where

N_{last}^{Feas} is the number of selected feasible test cases in last generation, and

N_{first}^{Feas} is the number of selected feasible test cases in first generation.

similarly the improvement factor of infeasible test cases (IF_I) is computed as

$$IF_I = \frac{N_{first}^{Unfeas} - N_{last}^{Unfeas}}{N_{first}^{Unfeas}} \quad (6.8)$$

where

N_{last}^{Unfeas} is the number of selected infeasible test cases in last generation, and

N_{first}^{Unfeas} is the number of selected infeasible test cases in first generation.

Table 6.5 summarizes the improvement of feasible test cases for different values of α and β which are based upon the results obtained from experiments discussed above. It is clear from Figure 6.10 which represents Table 6.5 graphically, it can be observed that highest improvement in feasible test cases is obtained for α=1.4 and β=70. Similarly from Figure 6.11 it is observed that for the same combination of α and β maximum improvement is obtained for infeasible test cases also. This is obvious because if number of feasible test cases increases then number of infeasible test cases will reduce and this improves overall generation.

Table 6.2 Results obtained using different combinations of α and β parameters

	α=1.2, β=50			α=1.2, β=70			α=1.2, β=90		
Gen#	**Feasible**	**Infeasible**	**Coverage**	**Feasible**	**Infeasible**	**Coverage**	**Feasible**	**Infeasible**	**Coverage**
10	71	129	22.73	72	128	18.18	51	149	18.18
20	133	67	31.82	101	99	18.18	123	77	27.27
30	124	76	31.82	84	116	18.18	94	106	36.36
40	114	86	31.82	63	137	18.18	95	105	45.45
50	102	98	40.91	77	123	27.27	82	118	50.00
60	93	107	40.91	82	118	31.82	88	112	59.09
70	104	96	45.45	117	83	40.91	84	116	63.64
80	115	85	45.45	62	138	50.00	66	134	63.64
90	108	92	45.45	103	97	59.09	102	98	68.18
100	135	65	50.00	113	87	63.64	81	119	68.18
110	127	73	59.09	115	85	72.73	105	95	72.73
120	128	72	63.64	137	63	77.27	83	117	72.73
130	121	79	72.73	134	66	77.27	106	94	81.82

	α=1.2, β=50			α=1.2, β=70			α=1.2, β=90		
Gen#	**Feasible**	**Infeasible**	**Coverage**	**Feasible**	**Infeasible**	**Coverage**	**Feasible**	**Infeasible**	**Coverage**
140	140	60	77.27	171	29	81.82	111	89	81.82
150	146	54	86.36	160	40	90.91	87	113	86.36
160	147	53	86.36	144	56	100.00	119	81	86.36
170	123	77	95.45				102	98	90.91
180	159	41	95.45				114	86	95.45
190	153	47	95.45				125	75	100.00
200	192	8	100.00						
	Feasible : 1.70 Infeasible: 0.94			**Feasible : 1.0 Infeasible: 0.56**			**Feasible : 1.45 Infeasible: 0.49**		

Table 6.3 Results obtained using different combinations of α and β parameters

	α=0.8, β=50			α=0.8, β=70			α=0.8, β=90		
Gen#	**Feasible**	**Infeasible**	**Coverage**	**Feasible**	**Infeasible**	**Coverage**	**Feasible**	**Infeasible**	**Coverage**
10	71	129	13.64	91	109	22.73	101	99	22.73
20	82	118	13.64	83	117	27.27	143	57	31.82
30	131	69	22.73	71	129	27.27	133	67	40.91
40	66	134	27.27	86	114	27.27	164	36	40.91
50	83	117	27.27	116	84	36.36	132	68	40.91
60	91	109	27.27	51	149	36.36	145	55	40.91
70	96	104	31.82	102	98	45.45	120	80	45.45
80	131	69	36.36	104	96	50	102	98	45.45
90	104	96	40.91	96	104	59.09	101	99	54.55
100	111	89	45.45	102	98	59.09	128	72	54.55
110	123	77	50	134	66	63.64	83	117	54.55
120	124	76	54.55	126	74	68.18	102	98	59.09
130	125	75	54.55	132	68	68.18	78	122	68.18
140	121	79	63.64	122	78	77.27	111	89	72.73
150	165	35	68.18	91	109	81.82	113	87	72.73
160	131	69	77.27	152	48	81.82	135	65	72.73
170	150	50	86.36	103	97	81.82	155	45	81.82
180	113	87	90.91	141	59	81.82	142	58	86.36
190	162	38	95.45	183	17	90.91	104	96	86.36
200	168	32	100	135	65	90.91	121	79	95.45
	Feasible : 1.36 Infeasible: 0.75			**Feasible : 0.48 Infeasible: 0.40**			**Feasible : 0.19 Infeasible: 0.20**		

Table 6.4 Results obtained using different combinations of α and β parameters

	α=1.4, β=50			α=1.4, β=70			α=1.4, β=90		
Gen#	**Feasible**	**Infeasible**	**Coverage**	**Feasible**	**Infeasible**	**Coverage**	**Feasible**	**Infeasible**	**Coverage**
10	82	118	22.73	62	138	18.18	63	137	13.64
20	102	98	27.27	103	97	18.18	132	68	18.18
30	123	77	27.27	102	98	18.18	84	116	27.27
40	122	78	31.82	130	70	18.18	86	114	31.82
50	95	105	31.82	121	79	27.27	102	98	31.82
60	82	118	40.91	101	99	36.36	104	96	36.36
70	84	116	40.91	132	68	36.36	124	76	36.36
80	102	98	45.45	115	85	36.36	125	75	45.45

	α=1.4, β=50			α=1.4, β=70			α=1.4, β=90		
Gen#	**Feasible**	**Infeasible**	**Coverage**	**Feasible**	**Infeasible**	**Coverage**	**Feasible**	**Infeasible**	**Coverage**
90	91	109	54.55	111	89	45.45	107	93	45.45
100	127	73	59.09	183	17	45.45	111	89	50
110	135	65	63.64	114	86	50	160	40	54.55
120	120	80	72.73	125	75	50	132	68	63.64
130	125	75	81.82	110	90	54.55	124	76	63.64
140	154	46	86.36	82	118	63.64	145	55	68.18
150	121	79	90.91	104	96	68.18	123	77	77.27
160	131	69	90.91	151	49	72.73	182	18	77.27
170	174	26	95.45	170	30	81.82	191	9	81.82
180	162	38	100	141	59	90.91	145	55	81.82
190				143	57	95.45	143	57	81.82
200				192	8	100	146	54	86.36
	Feasible : 0.98 **Infeasible: 0.67**			**Feasible : 2.09** **Infeasible: 0.94**			**Feasible : 1.32** **Infeasible: 0.60**		

Table 6.5 Improvement factor for feasible test cases (IF_F)

β \ α	0.8	1.2	1.4
50	1.36	1.70	0.98
70	0.48	1	2.09
90	0.19	1.45	1.32

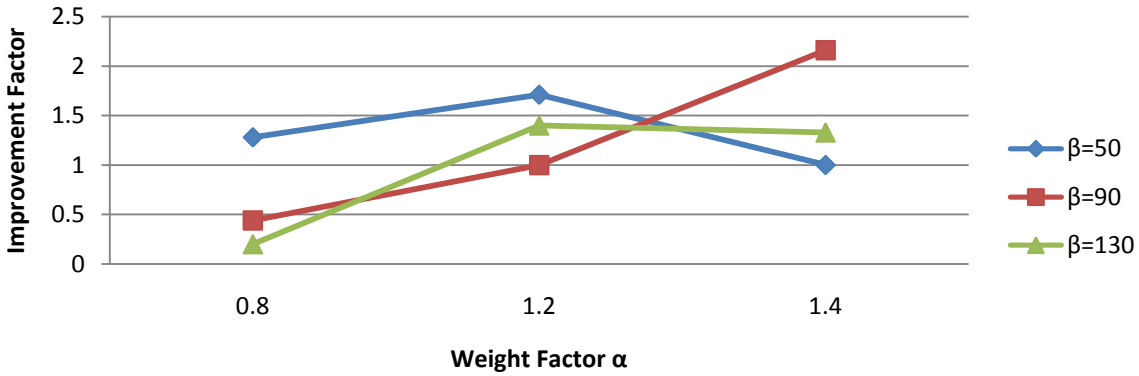


Figure 6.10 Variation of Improvement factor for feasible test cases

Table 6.6 Improvement factor for infeasible test cases (IF_I)

α \ β	0.8	1.2	1.4
50	0.75	0.94	0.67
70	0.40	0.56	0.94
90	0.2	0.49	0.60

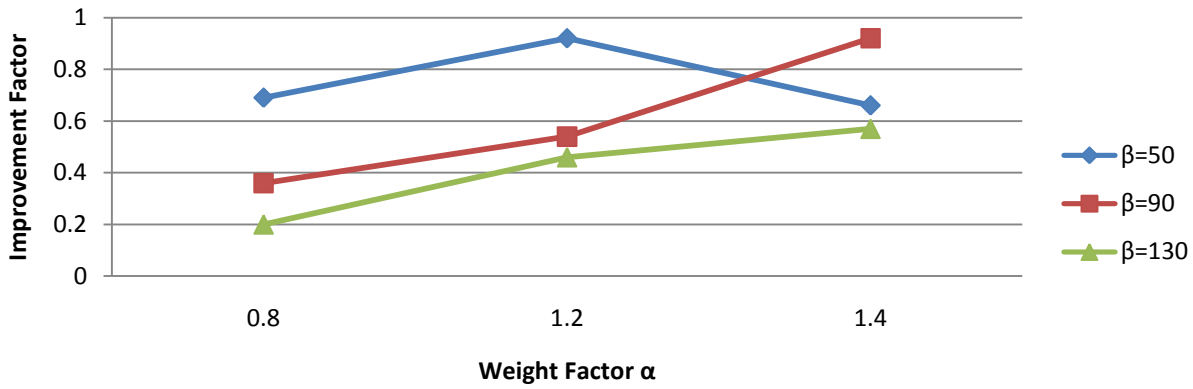


Figure 6.11 Variation of Improvement factor for infeasible test case

6.5.2. Other Case Studies

To validate the approach more case studies have been performed. Some selected methods from the four classes in java library are identified for the analysis. These four classes are ArrayList, HashMap, LinkedList and HashTable classes. These classes are chosen from util package of the standard Java library. Various methods from these classes which are considered for Method Under Test (MUT) are shown in table 6.7. For each MUT in these classes the experiments are performed as done in case study 1. The final result of all these experiments is tabulated in table 6.7 which shows values of CR, MR, α and β for which maximum improvement obtained through our improved approach as compared to exiting approach [Carlos et al. 2008]. As with the earlier case study 1, for these case studies there are 200 individuals per generation. From the table 6.7 we observe that there are 9 methods out of 16 for which the value of β is 90. For 4 methods β is 70. There are 2 methods which have β=50 and only 1 method with β=120. In our formulation β decides how many infeasible test cases will be considered for next generation by increasing their fitness value. Similarly α is responsible for keeping the weight of nodes at reasonable value for each test case execution. It has been observed that for those methods which have complex structure of CFG, the value of α is high.

Table 6.7 Results obtained for various methods of classes from Java util package

Class/Method	CR	MR	**α**	**β**	IF_F		IF_I	
ArrayList Class					**Existing**	**Improved**	**Existing**	**Improved**
trimToSize()	0.8	0.1	1.2	50	0.60	0.64	0.64	0.78
ensureCapacity(int)	0.6	0.2	1.2	70	0.63	0.58	0.71	0.88
indexOf(Object)	0.7	0.2	1.4	70	0.70	0.89	0.72	0.73
remove(int)	0.8	0.3	1	70	0.58	0.60	0.55	0.60

Class/Method	CR	MR	α	β	IF_F		IF_I	
remove(Object)	0.7	0.3	1.3	90	0.78	1.00	0.53	0.67
HashMap Class								
containsValue(Object)	0.7	0.2	1.4	90	0.62	0.55	0.98	0.67
removeMapping(Object)	0.6	0.3	1.4	70	0.53	0.40	0.67	0.40
LinkedList Class								
remove(Object)	0.7	0.3	1.2	90	0.91	1.43	0.59	0.77
clear()	0.6	0.3	1	50	0.56	0.44	0.74	0.36
indexOf(Object)	0.8	0.2	1.2	70	0.83	1.25	0.53	0.83
lastIndexOf(object)	0.7	0.3	1.4	50	0.75	1.00	0.79	0.82
removeLastOccurence(Object)	0.8	0.1	1.2	90	0.93	1.57	0.76	0.85
HashTable Class								
contains(Object)	0.7	0.1	1	50	1.40	1.50	0.61	0.64
containsKey(Object)	0.8	0.2	1.2	70	0.65	1.13	0.72	0.75
get(Object)	0.6	0.1	1.2	70	0.67	0.50	0.81	0.75
remove(Object)	0.7	0.2	1.4	90	0.51	0.42	0.71	0.63

Table 6.7 has been graphically represented in Figure 6.12. It can be observed that for most of the case studies 10 out of 16, the improvement factor is greater in our improved approach as compared to existing approach by Carlos et al. [2008].

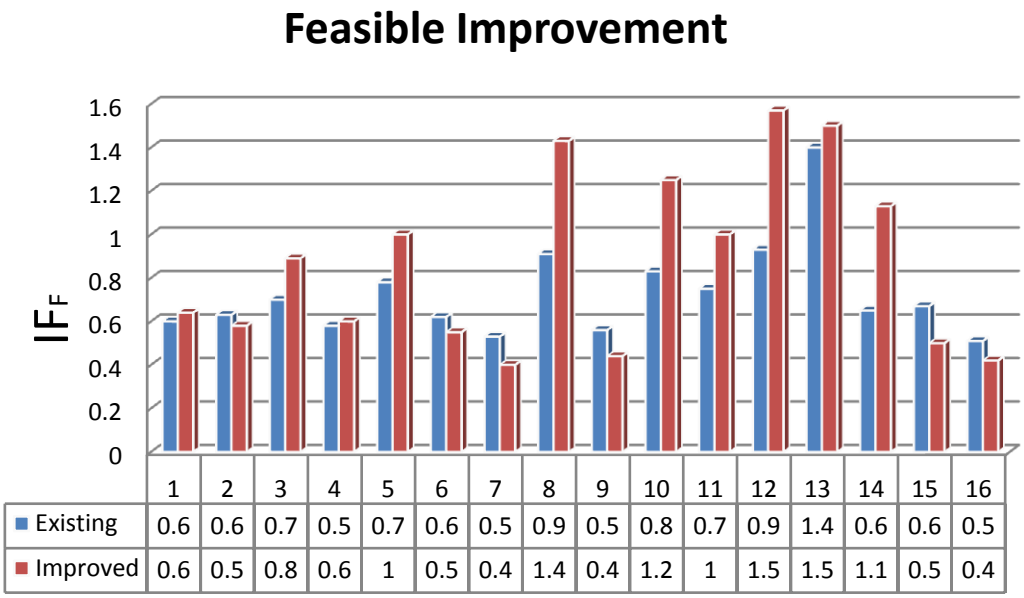


Figure 6.12 Improvement for feasible test cases in different case studies

Similarily, Figure 6.13 represents the improvements obtained for infeasible test cases in various case studies. Here also 11 out of 16 case studies have a greater improvement factor as compared to existing approach.

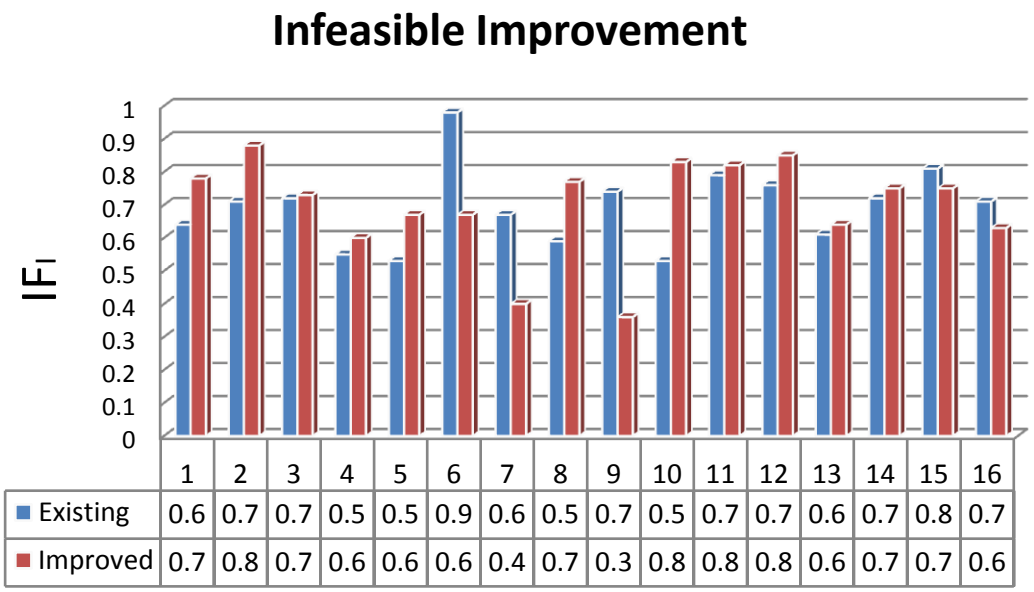


Figure 6.13 Improvement for infeasible test cases in different case studies

These results support our assumption described in section 6.4 that dependencies within loop iterations cause problems while guiding the fitness function and must be solved during structural testing.

6.6 Summary

In this chapter a technique of test case generation for object oriented software class has been discussed by applying genetic algorithms. The simplicity involved in unit testing using genetic algorithm is its greatest merit. During each of the iteration of the genetic algorithm a generation of individuals is generated. The iterations performed in the algorithm should be limited so that the computation time may not rise indefinitely. The problem may be because of this limited number of generations the solution obtained by genetic algorithms might be trapped in a local optimua and due to this it may fail to locate the required global optima. The test cases generated using such method may be trapped around some unwanted paths and in this manner may fail to find the new and interesting paths but when the distribution of test cases of the first generations are normally distributed then the possibility of being trapped in local optima is very low.

This research with improved strategy shows that genetic algorithms are useful in reducing the number of infeasible test cases by generating test cases for object oriented software. Furthermore, we improved genetic algorithm for structural testing and for generating more suitable test cases. Path testing weight reevaluation strategy is employed to develop infeasible test cases into feasible test cases at the later generations.

CHAPTER 7 *Conclusions and Suggestions for Future Research*

This chapter concludes the thesis by summarizing the achievements of this work, conclusions, limitations and giving directions for future work.

7.1 Summary of Achievements

The major achievement of this research is a novel and object oriented metric for maintenance relating it to indirect coupling and another is the development of an approach to automatically generate object-oriented unit tests.

On reviewing the state of the art in the research on coupling metrics, it is concluded that there are inherent ambiguities with many of the definitions of the metrics and thus the metrics cannot be measured objectively. Also a form of coupling that has not been explored very well — indirect coupling has been identified and argued that its potential impact on maintenance is significant and thus is worthy of study. To gather evidence in support of this and to find this relationship a methodology based on the philosophy of software measurement by key researchers is followed. This research idntifies the internal attributes of object oriented software that comprises indirect coupling and relates it with maintainability.

Also the algorithm presented in chapter 5 (Figure 5.5) provides solution in complex and seemingly infeasible situations. The specific sub-characteristic of maintainability of interest is the effort to trace the indirect coupling connections for the purposes of understanding and modification. The metric is applied to nine open source Java softwares to validate the proposed metrics. The initial results help significantly towards achieving a refined model of the relationship between indirect coupling and maintainability.

Statistical analysis is employed in this study and simple linear regression is the dominating statistical tool used for validation.

The results of the analyses of the nineobject oriented softwares show that:

1. There is a strong relationship between the proposed metrics and the maintenability in the object oriented software.
2. The metrics in combinations which are collected from the source code are able to predict the maintenance effort in software development.
3. The correlation is successfully validated.

In chapter 6 of this thesis genetic algorithm are applied to generate test cases for object oriented class units. Nine different combinations of weight factor α and infeasibility factor β and for each combination 20 generations or until full coverage are obtained. The existing work in this area is limited and this motivated for the exploration of the new approach. Because of these limitations such as the applicability of the existing approaches (e.g. able to handle only a particular type of classes effectively), in terms of achievable code coverage (suboptimal value) and in terms of maintainability of results, this research suggests the evolutionary class testing approach to address such limitations.

The class testing approach used in evolutionary testing is basically a search based approach which is used to generate the test sequences which may use class type arguments for method calls for object oriented programs. The test sequences generated in this way include calls to the public methods of the classes on which they operate upon and they do not try to access non-public methods and thus possibility of breaking encapsulation is avoided. These causes the tests to be more maintainable because applying any refractoring involving non public methods do not break their integrity.

The test sequences which do not lead to generate an executable test sequence are rarely defined by the representation which is used in this work to encode the test sequences for allowing evolutionary search. Call dependencies which are very significant to be preserved are kept as required as the test sequences are encoded as method call trees which regard the call depencies among the methods. This representation also ensures that the genetic operators like mutation and crossover do not disturb the executability of test sequences. Therefore, the check and corrections in representation to maintain the test sequences does not required at all.

The object oriented programs have state problem [McMinn 2005] therefore testing them requires the definition of methologies which could enhance the coverage of problematic structures and those paths of control flow graph which have less probability to be covered. In this reference, this thesis proposes the method to handle this problem by defining weighted control flow graph nodes and adapting the weight of nodes constantly in the search direction. This strategy also causes the fitness of feasible Test cases to fluctuate throughout the search process, and allows infeasibletest test sequences to be considered at certain points of the evolutionary search – namely, once the feasible test cases are no more interesting because they exercise recurrently traversed paths through the structure.

The approach presented in this research enables the application of an established and well proven genetic programming approach. The test sequences which are chosen to be represented by method call trees enable the application of other genetic programming techniques. Therefore in combination of other tools the presented technique may be used as an evolutionary test sequence generator. The presented approach basically is more general in nature and does not depend upon any particular genetic programming technique therefore; it is open for further improvements and other ideas to be incorporated from the field of evolutionary computation.

An empirical investigation in the effectiveness of the approach shows that it can perform well when the various parameters are set properly. The approach has been tested over various standard methods of java classes, giving effective results. Therefore, a significant contribution of this work is in proposing an approach which can generate arbitrary test sequence which is highly feasible for some given set of classes. These generated test sequences are very interesting while considering in the context of various other testing techniques.

The test case generation strategy discussed is validated for four classes which are taken from Java Utilities library. Different combination of mutation rate and crossover rates are taken for finding the average number of generations which are attaining full structural coverage as well as percentage of runs which are attaining full structural coverage in different generations. It has been observed that maximum coverage has been obtained for smaller value of mutation rate and average value of crossover rate. The research presented in this regard is a novel approach for the automatic unit test

generation tailored to object oriented systems. In particular, the proposed method tackles the test-generation problem as a search problem, solved using a holistic evolutionary algorithm.

7.2 Conclusions

The presented research addresses the issue of software quality measurement and test generation for its improvement. The main contributions of this research are twofold: at first we define an approach for determining the software quality through maintainability metrics by considering indirect form of coupling into account. Secondly it explores evolutionary testing techniques to be applied for test data generation. In this regard, this research focuses on a more effective fitness evaluation technique which can be used to generate better test cases.

Indirect form of coupling helps to address some of the problems in the status of coupling research. One problem is that the existing definitions of coupling do not capture the full essence of the original notion posed by Yourdon and Constantine [1979]. The indirect coupling form is one of them. The other problem is that the relationship between coupling (mostly Coupling Between Objects) and quality has not been explored much beyond statistical correlations, thus the presented approach toward establishing a precise relationship between indirect coupling and maintainability for better understanding and prediction presents a viable blueprint for research on other coupling metrics, direct or indirect.

This research summarizes a technique using Evolutionary Testing which can generate test sequences which are able to create arbitrary objects which may further be used to serve as arguments for succeeding methods. The presented approach can be used to generate automatic test cases where method parameters can accept primitive as well as object type arguments. The presented approach also proposes the technique to evaluate fitness of feasible as well as infeasible test cases, which provides sufficient guidance even in presence of runtime exceptions. The proposed technique is applied to various case studies involving methods from different classes of Java util library. The research shows that by selecting the defined parameters appropriately one can maximize the improvement of feasible as well as infeasible test cases also maximize the coverage. In our study for triangle classification program a highest value of improvement factor 2.16 is obtained for feasible

test cases and 0.92 is obtained for infeasible test cases. Similarly for other classes selected from util package of standard java library a highest value of improvement factor 1.57 is obtained for feasible test cases and 0.88 is obtained for infeasible test cases. The results obtained are compared with the results using existing technique and it was found that the technique proposed in this research gives much better results as compared to existing technique.

7.3 Limitations

Despite the promisingresults obtained, there are several points to improve the implemented evolutionary strategy. From the stand point of using an evolutionary optimization technique, which is the case of GAs to solve problems in evolutionary testing, in which case is tomaximize coverage of tests automatically generated, the aim here has been reached. Despite this, GAs in software testing have already been done previously by various researchers, the approach utilized here is different and equally efficient, considering the studies in Chapter6. But thinking more on the side of the evolutionary testing, from the stand point of evolutionary technique improvements are identified to be added to the presented approach. These improvements are explained below.

1. The study of metrics in chapter 5 must be expanded to cover a more representative sample of software systems. This includes non-open-source and commercial systems, and systems written in programming languages other than Java.
2. Indirect coupling forms other than simple transitive coupling should also be explored to further close the gap in the existing coupling definitions in the research field. For example when two classes A and B are coupled through a similar mechanism, except that in the middle there is persistency or a remote communication involved. This would be more difficult to systematically define and automatically detect through a tool, but this presents a significant burden to maintenance.
3. The presented approach requires a deeper analysis of the methods of the class being tested, in addition to knowing which object to instantiate, and know what methods to call the dependent object to certain points in the code are met (i.e., leave the object in the desired state for the preconditions of the test are made).

4. The strategy based on GA presented in chapter 6 is dependent upon structure of the MUT. The parameters α and β must be fine tuned to get the maximum improvement in test cases. The values of parameters α and β can not be generalized for all software.

7.4 Suggestions for future work

There is a high correlation for smaller path lengths, however the accuracy for predicting maintenance effort more experiments are required. The testing and structural coverage of non-public methods (via an object's public interface) is an important issue. Therefore, the future work may address the software testing challenges faced because of core principles of object orientation based upon concepts of abstraction, encapsulation and polymorphism.

The researchers in software testing area are yet studying the importance of abstraction and polymorphism in development of object oriented software and the ways in which it impacts the software testing process. Polymorphism concept refers to the ability of a variable or object type to take on multiple forms. Therefore the object oriented language having this feature allows the developers to develop programs more in general rather than being specific.

Our technique to compute indirect coupling relations takes into account static dependencies only. Implicit dependencies due to dynamic dependencies are not considered. Because of this the independent changes may be overestimated. Therefore in future work it would be interesting to explore dynamic dependencies as well as other types of dependencies also such as dependencies based on data flow and due to inheritance.

Our work considers the first level of method call dependencies, i.e. the indirect dependencies due to direct form of dependencies. We expect that the dependencies due to further levels are smaller due to information hiding. Therefore, such dependencies could be explored in future work.
Further the case studies were performed on various open source projects, therefore to generalize the findings of this study to encompass commercial projects; therefore the study over various commercial and other open source projects can be performed in future.

Future research may also involve addressing the oracle generation problem, and investigating the possibility of automating a mechanism for checking if the output of a program is correct given some input; in fact, the frequent non-existence of an oracle threatens to undo much of the progress made in automating Test Data generation, as a human tester is still required to perform this task manually.

LIST of *References*

1. [Abreu and Carapuca 1994] F.Abreu F. and Carapuca R., "Object-oriented software engineering: Measuring and controlling the development process", In Proceedings of the 4th International Conference on Software Quality, 1994.
2. [Abreu and Fernando 1995] Abreu F., Fernando B. "Design metrics for OO software system", ECOOP'95, Quantitative Methods Workshop, 1995.
3. [Abreu et al. 1995] Abreu F., Goulão M., and Esteves R., "Toward the Design Quality Evaluation of Object-Oriented Software Systems," Proceedings of Fifth International Conference on Software Quality, Austin, Texas, Oct. 1995.
4. [Alander et al. 1998] Alander J., Mantere T. and Turunen P. "Genetic Algorithm Based Software Testing, in Artificial Neural Nets and Genetic Algorithms", Springer-Verlag, Wien, Austria, pp. 325-328, 1998.
5. [Alexander and Offutt 1999] Alexander R. and Offutt A. "Analysis techniques for testing polymorphic relationships", Thirtieth International Conference on Technology of Object-Oriented Languages and Systems(TOOLS30), Santa Barbara, CA, pp. 104–114, 1999.
6. [Ammann and Offutt 2008] Ammann P. and Offutt J. "Introduction to Software Testing", Cambridge University Press, Cambridge, UK, 2008.
7. [Arisholm 2002] Arisholm E. "Dynamic Coupling Measures for Object-Oriented Software", Proceedings of the 8th International Symposium on Software Metrics (METRICS '02), IEEE Computer Society, Washington, DC, USA, pp. 33-42, 2002.
8. [Baig 2004] Baig I., "Measuring Cohesion and Coupling of Object-Oriented Systems", School of Engineering Blekinge Institute of Technology, Thesis No: MSE-2004:29, pp. 20-26, August 2004.
9. [Baker et al. 1990] Baker A., Bieman J., Fenton A., Gustafson D., Melton A. and Whitty R. "A philosophy for software measurement", Journal of Systems and Software, vol. 12, no. 3, pp. 389 – 416, July 1990.
10. [Bansiya 2002] Bansiya J. "A Hierarchical Model for object-oriented Design Quality Assessment", IEEE Transaction on Software Engineering, vol.28, no.1, 2002.
11. [Baresel et al. 2002] Baresel A., Sthamer H. and Schmidt M. "Fitness function design to improve evolutionary structural testing", Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002), New York, USA, pp. 1329–1336, 2002.
12. [Basili and Hutchens 1983] Basili V. and Hutchens D., "An Empirical Study of a Syntactic Complexity Family", IEEE Transasctions on Software Engineering, vol. 9, no.6, pp.664-672, 1983.

13. [Basili et al. 1996] Basili V., Briand L. and Melo W. “A validation of object-oriented design metrics as quality indicators”, Software Engineering, IEEE Transactions on, vol.22, no.10, pp. 751-761, 1996.

14. [Berard 1994] Berard E. “Issues in the testing of object-oriented software”, Electro’94 International, IEEE Computer Society Press, pp. 211–219, 1994.

15. [Bernot et al. 1991] Bernot G., Gaudel M. and Marre B. “Software testing based on formal specifications: a theory and a tool”, Software Engineering Journal, vol. 6, no. 6, pp. 387-405, 1991.

16. [Bieman and Kang 1995] Bieman J. and Kang B., “Cohesion and Reuse in an Object-Oriented System”, Proceedings of ACM Symposium, Software Reusability (SSR’94), pp.259-262, 1995.

17. [Beizer 1990] Beizer, B., “Software Testing Techniques”, 2nd edition. Boston, MA: International Thomson Computer Press.

18. [Beizer 1995] Beizer B., “Black-Box Testing: Techniques for Functional Testing of Software and Systems”, John Wiley & Sons, Inc., New York, NY, USA, 1995.

19. [Bilal and Black 2006] Bilal H. and Black S. “Computing ripple effect for object oriented software”, Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE), Nantes, France, July 2006.

20. [Bilow 1992] Bilow S. “Applying graph-theoretic analysis models to object oriented system models”, OOPSLA 92, Workshop on Metrics for Object Oriented Software Eng., Position Paper, 1992.

21. [Binder 1996] Binder R. “Testing object-oriented software: A survey”, Journal of Software Testing, Verification & Reliability, vol. 6, no. 3, pp. 125–252, 1996.

22. [Binkley and Schach 1998] Binkley A. and Schach S., “Validation of the coupling dependency metrics as a predictor of run time failures and maintainability measures”, Proceedings of 20th International conference of software engineering, pp. 452-455, 1998.

23. [Boehm 2003] Barry Boehm, “Value-Based Software Engineering,” Software Engineering Notes, vol. 28, no. 2, ACM, March 2003.

24. [Boehm et al. 1976] Boehm B., Brown J. and Lipow M. “Quantitative evaluation of software quality”, Proceedings of the 2nd international conference on Software engineering (ICSE '76), IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 592-605, 1976.

25. [Booch 1991] Booch G. “Object Oriented Design with Applications”, Benjamin-Cummings Publication Co., Inc., Redwood City, CA, USA.

26. [Booch et al. 1999] Booch G., Rumbaugh J. and Jacobson I., “The Unified Modeling Language User Guide”, Addison- Wesley, Rational Software Corporation, 1999.

27. [Bowen 1978] Bowen J., "Are Current Approaches Suffcient for Measuring Software Quality?" Proceedings of the ACM Software Quality Assurance Workshop, pp. 148-155, November 1978.

28. [Briand et al. 1997] Briand L., Devanbu P. and Melo W., “An Investigation into Coupling Measures for C++,” Proc. 19th Int’l Conf. Software Eng., ICSE‘97, Boston, pp. 412-421, May 1997.

29. [Briand et al. 1998] Briand L., Daly W. and Wust J. “Unified Framework for Cohesion Measurement in Object-Oriented Systems”, Empirical Software Engineering, vol. 3, pp.65-117, 1998.

30. [Briand et al. 1999] Briand L, Daly W. and Wust J. “A Unified Framework for Coupling Measurement in Object-Oriented Systems”, IEEE Transactions on software Engineering, vol. 25, no. 1, pp. 91-121, 1999.

31. [Briand et al. 2000] Briand L, Wust J. and Daly W. “Exploring the relationship between design measures and software quality in object-oriented systems”, Journal of Systems and Software, vol. 51, no. 3, pp. 245-273, 2000.

32. [Briand et al. 2002] Briand L., Feng J. and Labiche Y. “Using genetic algorithms and coupling measures to devise optimal integration test orders”, Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, Ischia, Italy, IEEE Computer Society Press, pp. 43-50, 2002.

33. [Card et al. 1986] Card D., Church V. and Agresti W. “An Empirical Study of Software Design Practices”, IEEE Transactions on Software Engineering, vol. 12, no. 2, 1986.

34. [Carlos et al. 2008] Carlos J., Ribeiro B., Rela Z. and Vega F. “A strategy for evaluating feasible and unfeasible test cases for the evolutionary testing of object-oriented software”, Proceedings of the 3rd international workshop on Automation of software test (AST '08), New York, NY, USA, ACM, pp.85-92, 2008.

35. [Cavano and McCall 1978] Cavano J. and McCall J. “A Framework for the Measurement of Software Quality”, Proceedings of ACM Software Quality Assurance Workshop, pp. 133–139, 1978.

36. [Chang 2000] Chang K. “Reusability and maintainability metrics for object-oriented software”, Proceedings of the 38th annual on Southeast regional conference (ACM-SE 38). ACM, New York, NY, USA, pp. 88-94, 2000.

37. [Cheatham and Mellinger 1990] Cheatham T. and Mellinger L. “Testing object-oriented software systems”, Proceedings of the Eighteenth Annual Computer Science Conference, ACM Press, New York, pp. 161–165, 1990.

38. [Chen et al. 1998] Chen H., Tse T., Chan F. and Chen T. “In black and white: An integrated approach to class-level testing”, ACM Transactions on Software Engineering and Methodology, vol. 7, no. 3, pp. 250–295, 1998.

39. [Chen et al. 2000] Chen H., Tse T. and Deng Y. “ROCS: An object-oriented class-level testing system based on the relevant observable contexts technique”, Information and Software Technology, vol. 42, no. 10, pp. 677–686, 2000.

40. [Cheng and Krishnakumar 1983] Cheng K. and Krishnakumar A. “Automatic functional test generation using the extended finite state machine model”, Proceedings of the 30th International Conference on Design Automation (Dallas, Texas, United States, June 14 - 18, 1993), DAC '93, ACM Press, New York, NY, pp. 86-91, 1983.

41. [Chidamber and Kemerer 1991] Chidamber S. and Kemerer C. “Towards a Metrics Suite for Object Oriented design”, Proceedings of Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA’91), Published in SIGPLAN Notices, vol. 26, no. 11, pp. 197-211, 1991.

42. [Chidamber and Kemerer 1994] Chidamber S. and Kemerer C. “A Metrics Suite for Object-Oriented Design”, IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476-493, 1994.

43. [Chowdhury and Zulkernine 2010] Chowdhury I. and Zulkernine M. “Using complexity, coupling, and cohesion metrics as early predictors of vulnerabilities”, Journal of Systems Architecture, vol. 57, no. 3, pp. 294-313, 2010.

44. [Churcher and Shepperd 1995] Churcher N. and Shepperd M., “Towards a Conceptual Framework for Object Oriented Software Metrics,” Software Engineering Notes, vol. 20, no. 2, pp. 69-76, 1995

45. [Coad and Yourdon 1991] Coad P. and Yourdon E. “Object-Oriented Design”, Yourdon Press, Upper Saddle River, NJ, USA.

46. [Coleman et al. 1994] Coleman D., Ash D., Lowther B. and Oman P., “Using Metrics to Evaluate Software System Maintainability”, IEEE Computer, vol. 27, no. 8, pp. 44-49. 1994.

47. [Coleman et al. 1995] Coleman D., Lowther B. and Oman P. “The Application of Software Maintainability Models in Industrial Software Systems”, Journal of Systems and Software, vol. 29, no. 1, pp. 3-16, 1995.

48. [Cohen 1988] Cohen J. “Statistical Power Analysis for the Behavioral Sciences”, 2nd edition, Hillsdale NJ: Lawrence Erlbaum, 1988.

49. [Coplien 1993] Coplien J., “Looking over one’s shoulder at a c++ program”, AT&T Bell Labs, Tech. Memo., Jan. 1993.

50. [Cornett 2002] Cornett S., “Code Coverage Analysis”, Bullseye Testing Technology 2002, available at: http://www.bullseye.com/coverage.html

51. [Dagpinar and Jahnke 2003] Dagpinar M. and Jahnke J. “Predicting Maintainability with Object-Oriented Metrics - An Empirical Comparison”, 10th Working Conference on Reverse Engineering, British Columbia, Canada, 2003.

52. [Deason et al. 1991] Deason W., Brown D., Chang K. and Cross J. “A Rule-Based Software Test Data Generator”, IEEE Trans. on Knowledge and Data Engineering, vol. 3, no. 1, pp. 108-117, 1991.

53. [Dejong 1975] Dojong, K. “Analysis of the behavior of a class of genetic adaptive systems”, Ph. D. thesis, University of Michigan, Ann Arbor, 1975.

54. [DeMillo and Offutt 1991] DeMillo R. and Offutt A., “Constraint-Based Automatic Test Data Generation”, IEEE Transactions on Software Enginering, vol. 17, no. 9, pp. 900-910, 1991.

55. [Dromey 1995] Dromey G. “A Model for Software Product Quality”, IEEE Transactions on Software Engineering, vol. 21, no. 2, pp. 146-162, 1995.

56. [Duda et al. 2000] Duda, R., Hart, P. and Stork D., "Pattern Classification" (2nd Edition), Wiley-Interscience, pp. 433 – 439, 2000.

57. [Eder et al. 1994] Eder J., Kappel G. and Schrefl M., "Coupling and Cohesion in Object-Oriented Systems," Technical Report, Univ. of Klagenfurt, 1994.

58. [Erdil 2003] Erdil K., Finn E., Keating K., Meattle J., Park S. and Yoon D. "Software Maintenance As Part of the Software Life Cycle", Comp180, Software Engineering Project, December 16, 2003.

59. [Fenton 1991] Fenton E. "Software Metrics: A Rigorous Approach", International Thomson Computer Press, Boston, MA, USA.

60. [Fenton 1994] Fenton N. "Software measurement: A necessary scientific basis", IEEE Transactions on Software Engineering, vol. 20, no. 3, pp. 199 – 206, March 1994.

61. [Fenton and Neil 1999] Fenton N. and Neil M. "A Critique of Software Defect Prediction Models", IEEE Transactions on Software Engineering, vol. 25, no. 5, pp. 675-689, 1999.

62. [Fenton and Pfleeger 1998] Fenton N. and Pfleeger S. "Software Metrics - A Rigorous & Practical Approach", 2nd Edition, PWS Pub. Co., Boston, MA, USA, 1998.

63. [Ferguson and Korel 1996] Ferguson R. and Korel B. "The chaining approach for software test data generation", ACM Transactions on Software Engineering and Methodology, vol. 5, no. 1, pp. 63-86, 1996.

64. [Ferrer et al. 2012] Ferrer J., Kruse P., Chicano F. and Alba E., "Evolutionary algorithm for prioritized pairwise test data generation", In Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference (GECCO '12), Terence Soule (Ed.). ACM, New York, NY, USA, pp. 1213-1220, 2012.

65. [Firesmith 1993] Firesmith D. "Testing object-oriented software", Eleventh International Conference on Technology of Object-Oriented Languages and Systems, Prentice-Hall, Englewood Cliffs, New Jersey, pp. 407-426, 1993.

66. [Frappier et al. 1994] Frappier M., Matwin S. and Mili A. "Maintainability: Factors and Criteria", Software Metrics Study, Tech. Memo. 1, Canadian Space Agency, St-Hubert, Canada, 1994.

67. [Friedman and Voas 1995] Friedman M. and Voas J. "Software Assessment: Reliability, Safety and Testability", John Wiley and Sons, 1995.

68. [Friedman et al. 2002] Friedman G., Hartman A., Nagin K., and Shiran T. "Projected state machine coverage for software testing", Proceedings of the 2002 ACM SIGSOFT international Symposium on Software Testing and Analysis (Roma, Italy, July 22 - 24, 2002), ISSTA '02. ACM Press, New York, NY, pp. 134-143, 2002.

69. [Gallagher and Offutt 2004] Gallagher L. and Offutt J. "Integration Testing of Object-oriented Components Using FSMS: Theory and Experimental Details", Technical report ISE-TR-04-04, Department of Information and Software Engineering, George Mason University, Fairfax, VA, July 2004.

70. [Gao et al. 2003] Gao J., Tsao J. and Wu Y. "Testing and Quality Assurance for Component-Based Software", USA: Artech House, 2003.

71. [Garvin 1984] Garvin D., “What does ‘product quality’ really mean?” Sloan Management Review, vol. 26, pp. 25–45.

72. [Genero et al. 2001] Genero M., Olivas J., Piattini M. and Romero F. “Using metrics to predict OO information systems maintainability”, Proceedings of the 13th International Conference on Advanced Information Systems Engineering, Springer-Verlag, London, UK, pp. 388-401, 2001.

73. [Genero et al. 2003] Genero M., Piattini M., Manso E. and Cantone G. “Building UML class diagram maintainability prediction models based on early metrics”, Proceedings of the 9th International Symposium on Software Metrics (Metrics 2003), IEEE Computer Society, Sidney, Australia, pp. 263–275, 2003.

74. [Godfrey and German 2008] Godfrey M., German D. “The past, present, and future of software evolution”, Frontiers of Software Maintenance, FoSM 2008, pp.129-138, 2008.

75. [Gross and Mayer 2002] Gross H. and Mayer N. “Evolutionary Testing in Component-Based Real-Time System Construction”, Genetic and Evolutionary Computation Conference (GECCO), Search-based Software Engineering Track, New York, N.Y., pp. 207-214, 2002.

76. [Gu et al. 1994] Gu D., Zhong Y. and Ali S. “On testing of classes in object-oriented programs”, Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research (CASCON '94), John Botsford, Ann Gawman, Morven Gentleman, Evelyn Kidd, Kelly Lyons, Jacob Slonim, and Howard Johnson (Eds.), IBM Press, pp. 22-30, 1994.

77. [Gui and Scott 2006] Gui G. and Scott P. “Coupling and Cohesion Measures for Evaluation of Component Reusability”, Proceedings of International Workshop on Mining Software Repositories, Shanghai, China, pp. 18-21, 2006.

78. [Gulezian 1991] Gulezian R. “Reformulating and calibrating COCOMO”, Journal of Systems and Software, vol. 16, no. 3, pp. 235–242, 1991.

79. [Gupta and Rohil 2012] Gupta N. and Rohil M. “Exploring Possibilities of Reducing Maintenance Effort in Object Oriented Software by Minimizing Indirect Coupling”, Proceedings of the Second International Conference on Computer Science, Engineering & Applications (ICCSEA 2012), May 25-27, 2012, New Delhi, India, Published as Book chapter in Advances in Computer Science, Engineering & Applications, Springer Berlin/Heidelberg, pp.959-965, 2012.

80. [Gupta and Saini 2008] Gupta N. and Saini D. “Class Level Test Case Generation in Object Oriented Software Testing”, International Journal of Information Technology and Web Engineering, IGI Publishing, Hershey, Pennsylvania, USA, vol. 3, no. 2, pp.19-26, 2008.

81. [Gupta et al. 1998] Gupta N., Mathur A. and Soffa A. “Automated test data generation using an iterative relaxation method”, SIGSOFT Software Engineering Notes, vol. 23, no. 6, pp. 231-244, 1998.

82. [Guttag et al. 1993] Guttag J., Horning J., Garland S., Jones K., Modet A. and Wing J. “Larch: Languages and Tools for Formal Specification Texts and Monographs”, Computer Science series Springer-Verlag, NY, 1993.

83. [Halstead 1975] Halstead H. “Elements of Software Science”, Elsevier Publications, N-Holland, 1975.

84. [Harman and Jones 2001] Harman M. and Jones B. “Search-based software engineering, Information & Software Technology”, vol. 43, no.14, pp. 833-839, 2001.

85. [Harman and McMinn 2007] Harman M. and McMinn P. “A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation”, Proceedings of the 2007 international symposium on Software testing and analysis (ISSTA '07), ACM, New York, NY, USA, pp. 73-83, 2007.

86. [Harrison et al. 1982] Harrison W., Magel K., Kluczny R. and DeDock A. “Applying Software Complexity Metrics to Program Maintenance”, IEEE Computer, vol. 15, no. 9, pp. 65-79, 1982.

87. [Harrison et al. 1998] Harrison R., Counsell S. and Nithi R. “An Evaluation of MOOD set of Object-Oriented Software Metrics”, IEEE Transactions on Software Engineering, vol. 24, no.6, pp. 491-496, 1998.

88. [Harrold and McGregor 1992] Harrold M., McGregor J. “Incremental testing of object-oriented class structures”, Proceedings of the 14th International Conference on Software Engineering (ICSE), Melbourne, Australia, IEEE Computer Society Press, Los Alamitos, CA, May 1992; pp. 68–80, 1992.

89. [Hayes et al. 2004] Hayes J., Patel S. and Zhao L. “A Metrics-Based Software Maintenance Effort Model,” Proceedings of 8th European Conference on Software Maintenance and Reengineering (CSMR'04), 24 – 26 Mar. 2004, IEEE Computer Society, pp. 254 – 258, 2004.

90. [Henry and Wake 1991] Henry S. and Wake S., “Predicting Maintainability with Software Quality Metrics,” Software Maintenance: Research and Practice, vol. 3, pp. 129-143, 1991.

91. [Hitz and Montazeri 1995] Hitz M. and Montazeri B. “Measuring coupling and cohesion in object-oriented systems”, Proceedings of International Symposium on Applied Corporate Computing, Monterrey, Mexico, 1995.

92. [Hoare 1972] Hoare C. “Proof of correctness of data representations”, Acta Informatica, vol. 1, no. 4, pp. 271-281, 1972.

93. [Hoffman and Strooper 1993] Hoffman D. and Strooper P., "Graph-based module testing", In Proceedings of the 16th Australian Computer Science Conference, pp. 479—487, February 1993.

94. [Holland 1975] Holland J. “Adaptation in Natural and Artificial Systems”, University of Michigan Press, Ann Arbor, 1975.

95. [Hopkins 2003] Hopkins W. “A New View of Statistics”, SportScience, Dunedin, New Zealand, 2003, available at: http://www.sportsci.org/resource/stats

96. [Howden 1982] Howden W. “Weak Mutation Testing and Completeness of Test Sets”, IEEE Transactions on Software Engineering, vol. 8, pp. 371-379, 1982.

97. [Hudli et al. 1994] Hudli R., Hoskins C. and Hudli A. “Software Metrics for Object Oriented Designs”, IEEE, 1994.

98. [IEEE 2006] IEEE, “IEEE Standard for Software Maintenance”, IEEE Std 14764- 2006. The Institute of Electrical and Electronics Engineers, Inc. 2006.

99. [ISO 2001] ISO 9126:2001 “Quality management systems - Fundamentals and vocabulary”, 2001.

100. [Inkumsah and Xie 2007] Inkumsah K. and Xie T. “Evacon: a framework for integrating evolutionary and concolic testing for object-oriented programs”, Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE '07), ACM, New York, NY, USA, pp. 425-428.

101. [Jones et al. 1996] Jones B. et al. “Automatic Structural Testing Using Genetic Algorithms”, Software Engineering Journal, vol. 11, no. 5, 1996.

102. [Jorgensen 2002] Jorgensen P., “Software Testing: A Craftmans approach”, 2nd Edition, CRC Press, USA, 2002.

103. [Kafura and Reddy 1987] Kafura D. and Reddy R. “The Use of Software Complexity Metrics in Software Maintenance”, IEEE Trans. Software Engineering, vol. SE-13, no. 3, pp. 335-343, 1987.

104. [Kataoka et al. 2002] Kataoka Y., Imai T., Andou H. and Fukaya T. “A Quantitative Evaluation of Maintainability Enhancement by Refactoring”, International Conference on Software Maintenance (ICSM'02), pp. 576-585, 2002.

105. [Kearney et al. 1986] Kearney J. et al., “Software complexity measurement,” Communications of the ACM, vol. 29, no. 11, pp. 1044-1050, 1986.

106. [Khoshgoftaar et al. 2000] Khoshgoftaar T., Shan R. and Allen E. “Improving Tree-Based Models of Software Quality with Principal Components Analysis”, Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE '00). IEEE Computer Society, Washington, DC, USA, pp. 198-209, 2000.

107. [Kim and Lerch 1991] Kim J. and Lerch J. “Cognitive processes in logical design: Comparing object-oriented and traditional functional decomposition software methodologies,” Working Paper, Camegie Mellon Univ. Graduate School of Industrial Admin., 1991.

108. [Kitchenham and Pfleeger 1996] Kitchenham B. and Pfleeger S. “Software Quality: The Elusive Target”, IEEE Software, vol. 13, no. 1, pp. 12-21, 1996.

109. [Koomen and Pol 1999] Koomen T. and Pol M. “Test Process Improvement—A Practical Step-by-Step Guide to Structured Testing”, Addison-Wesley, USA, 1999.

110. [Koten and Gray 2006] Koten C. and Gray A. “An Application of Bayesian Network for Predicting Object-Oriented Software Maintainability”, Information and Software Technology, vol. 48, no. 1, pp. 59 – 67, 2006.

111. [Koza 1992] Koza D. “Genetic Programming: On the Programming of Computers by Means of Natural Selection”, MIT Press, Cambridge, MA, 1992.

112. [Kung et al. 1995] Kung D., Gao J., Hsia P., Toyoshima Y., Chen C., Kim Y. and Song Y. “Developing an object-oriented software testing and maintenance environment”, Communications of the ACM, vol. 38, no. 10, pp. 75–87, 1995.

113. [Lacerda and Carvalho 1999] Lacerda E.and Carvalho, A. “Introduction to genetic algorithms”, XIX National Congress of the Brazilian Computer Society, Proceedings, vol. 2, p. 51-125, 1999.

114. [Lake and Cook 1994] Lake A. and Cook C. “Use of factor analysis to develop OOP software complexity metrics”, Proc. 6th Annual Oregon Workshop on Software Metrics, Silver Falls, Oregon, 1994.

115. [Lee 2007] Lee Y., “Automated Source Code Measurement Environment for Software Quality”, Dissertation, Auburn Univeristy, 2007.

116. [Lee et al. 1995] Lee Y., Liang B., Wu S. and Wang F., “Measuring the Coupling and Cohesion of an Object-Oriented Program Based on Information Flow,” Proceedings of International Conference on Software Quality, Maribor, Slovenia, 1995.

117. [Leino 2008] Leino M. “Specification and verification of object-oriented software”, Engineering Methods and Tools for Software Safety and Security, NATO Science for Peace and Security Series D: Information and Communication Security, IOS Press, vol. 22, pp. 231–266, 2009.

118. [Lewis and Wiener 1998] Lewis J. and Wiener R. “An Introduction to Object-oriented Programming and Smalltalk”, Addison- Wesley, pp. 49-60, 1988.

119. [Li and Henry 1993] Li W. and Henry S. “Object-oriented metrics that predict maintainability”, The Journal of Systems and Software, vol. 23, no. 2, pp. 111–122, 1993.

120. [Lieberherr et al. 1988] Lieberherr K., Holland I., and Riel A., “Object oriented programming: An objective sense of style”, Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA '88), Norman Meyrowitz (Ed.). ACM, New York, NY, USA, pp. 323-334.

121. [Lim et al. 2005] Lim J., Jeong S. and Schach S. “An empirical Investigation of the Impact of the Object-Oriented Paradigm on the Maintainability of Real-World Mission-Critical Software”, Journal of Systems and Software, vol. 77, no. 2, pp. 131-138, 2005.

122. [Lincke and Lowe 2006] Lincke R. and Lowe W. “Validation of a standard- and metric-based software quality model”, Proceedings of the 10th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2006), pp. 81-90, 2006.

123. [Liu and Khoshgoftaar 2003] Liu Y. and Khoshgoftaar T. “Building Decision Tree Software Quality Classification Models Using Genetic Programming”, Proccedings of the Genetic and Evolutionary Computation Conference (GECCO ’03), Springer, vol. LNCS-2724, pp. 1808-1809, 2003.

124. [Liu et al. 2005] Liu X., Wang B. and Liu H. “Evolutionary search in the context of object-oriented programs”, MIC2005, The Sixth Metaheuristics International Conference, September 2005.

125. [Lochmann 2010] Lochmann K. “Engineering Quality Requirements Using Quality Models”, Proceedings of the 2010 15th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '10). IEEE Computer Society, Washington, DC, USA, pp. 245-246, 2010.

126. [Lorenz and Kidd 1994] Lorenz M. and Kidd J. “Object-Oriented Software Metrics”, Prentice Hall, Upper Saddle River, New Jersey, USA, 1994.

127. [Mantero and Alander 2005] Mantere T. and Alander J. “Evolutionary software engineering, a review”, Applied Soft Computing, vol. 5, no. 3, pp. 315-331, 2005.

128. [Marinescu 2005] Marinescu R. “Measurement and Quality in Object-Oriented Design”, Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05). IEEE Computer Society, Washington, DC, USA, pp. 701-704, 2005.

129. [Mathur 2008] Mathur A. “Foundations of Software Testing”, Pearson Education, 2008.

130. [McCabe 1982] McCabe T. “Structured Testing”, IEEE Computer Society Press, Silver Spring, Maryland, 1982.

131. [McCall et al. 1977] McCall J., Richards P. and Walters G. “Factors in Software Quality”, vol. 1, 2, and 3, AD/A-049-014/015/055, National Tech. Information Service, 1977.

132. [McMinn 2003] McMinn P. and Holcombe M. “The state problem for evolutionary testing”, Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003), Lecture Notes in Computer Science 2274, Chicago, USA, Springer-Verlag, pp. 2488–2498, 2003.

133. [McMinn 2004] McMinn P. “Search-based test data generation: A survey”, Journal on Software Testing, Verification and Reliability, vol. 14, no. 2, pp. 105–156, 2004.

134. [McMinn 2005] McMinn P. “Evolutionary Search for Test Data in the Presence of State Behavior”, PhD Thesis, University of Sheffield, 2005.

135. [McMinn et al. 2009] McMinn P., Binkley D., Harman M. “Empirical evaluation of a nesting testability transformation for evolutionary testing”, ACM Trans Software Engineering and Methodology, vol. 18, no. 3, pp. 1–26, 2009.

136. [Melton 1996] A. Melton, editor, "Software Measurement", International Thomson Computer Press, 1996.

137. [Michael and McGraw 1998] Michael C. and McGraw G. “Automated software test data generation for complex programs”, Proceedings of IEEE International Conference on Automated Software Engineering (ASE’98), IEEE Computer Society, pp. 136–146, 1998.

138. [Misra 2005] Misra S. “Modelling Design/Coding Factors that Drive Maintainability of Software Systems”, Software Quality Journal, vol. 13, pp. 297-320, 2005.

139. [Montana 1995] Montana D. “Strongly typed genetic programming”, Evolutionary Computation, vol. 3, no. 2, pp. 199-230, 1995.

140. [Moreau and Dominick 1989] Moreau D. and Dominick W. “Object-Oriented Graphical Information Systems: Research Plan and Evaluation Metrics”, Journal of Systems and Software, vol.10, pp. 23-28, 1989.

141. [Morris 1988] Morris K., “Metrics for object oriented software development,” Masters thesis, M.I.T., Sloan School of Management, Cambridge, MA, 1988.

142. [Munson and Khoshgoftaar 1989] Munson J. and Khoshgoftaar T., "The Dimensionality of Program Complexity", Proceedings of the International Conference on Software Engineering, pp. 245-253, 1989.

143. [Murphy et al. 1994] Murphy G., Townsend P. and Wong P. “Experiences with cluster and class testing”, Communications of the ACM, vol. 37, no. 9, pp. 39-47, 1994.

144. [Myers 1979] Myers G. “The Art of Software Testing”, John Wiley and Sons, New York, 1979.

145. [Naik 1997] Naik K. “Efficient computation of unique input/output sequences in finite-state machines”, IEEE/ACM Transactions on Networking, vol. 5, no. 4, pp. 585-599, Aug. 1997.

146. [Offutt et al. 2008] Offutt J., Abdurazik A. and Schach S.R. “Quantitatively measuring object-oriented couplings”, Software Quality Control, vol. 16, no. 4, pp. 489-512, 2008.

147. [Oman and Hagemeister 1994] Oman P. and Hagemeister J. “Construction and Testing of Polynomials Predicting Software Maintainability”, Journal of Systems and Software, vol. 24, no. 3, pp. 251 – 266, 1994.

148. [OOTC 1993] IBM Object-Oriented Technology Council, “IBM Object-Orienied Metrics”, IBM internal technical paper, February 2, 1993.

149. [O'Regan 2002] O'Regan G. “A practical approach to software quality”, Springer, 1 edition, 2002.

150. [Orso and Pezze 1999] Orso A. and Pezze M. “Integration testing of procedural object-oriented languages with polymorphism”, Proceedings of the 6th International Conference on Testing Computer Software: Future Trends in Testing (TCS'99), Washington, DC, June 1999.

151. [Osterweil 1996] Osterweil L. “Strategic Directions in Software Quality”, ACM Computing Surveys, vol. 28, no. 4, pp. 738-750, 1996.

152. [Ostrand et al. 2005] Ostrand T., Weyuker E. and Bell R. “Predicting the location and number of faults in large software systems”, IEEE Transactions on Software Engineering, vol. 31, no. 4, pp. 340–355, 2005.

153. [Parnas 1972] Parnas D. “On the Criteria to be Used in Decomposing Systems into Modules”, Communications of the ACM, vol. 15, no. 12, Dec 1972.

154. [Pfleeger and Atlee 2006] Pfleeger S. and Atlee J. “Software Engineering -Theory and Practice”, Pearson and Prentice Hall, 2006.

155. [Pfleeger et al. 1990] Pfleeger S. and Palmer J. “Software estimation for object-oriented systems”, International Function Point Users Group Fall Conference, San Antonio, TX, 1990, pp. 181-196.

156. [Piszcz and Soule 2006] Piszcz A. and Soule T. “A survey of mutation techniques in genetic programming”, Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation (Seattle, Washington, USA, July 08 - 12, 2006). GECCO '06. ACM, New York, NY, pp. 951-952, 2006.

157. [Prather 1984] Prather R. “An axiomatic theory of software complexity measures”, The Computer Journal, vol. 27, no. 4, pp, 340-346, 1984.

158. [Pressman 2005] Pressman R. “Software Engineering: A Practitioner’s Approach”, 6th edition, McGraw-Hill, New York, 2005.

159. [Rajaraman and Lyu 1992] Rajaraman C., Lyu M. “Reliability and Maintainability Related Software Coupling Metrics in C++ Programs”, Proceedings 3rd IEEE International Symposium on Software Reliability Engineering (ISSRE'92), pp. 303-311, 1992.

160. [Rajlich and Bennett 2000] Rajlich T. and Bennett K. “A Staged Model for the Software Life Cycle”, Computer, vol. 33, no. 7, pp. 66-71, 2000.

161. [Rela 2004] Rela L. “Evolutionary computing in search-based software engineering”, Master’s thesis, Lappeenranta University of Technology, Department of Technology, 2004.

162. [Riaz et al. 2009] Riaz M., Mendes E. and Tempero E. “A Systematic Review of Software Maintainability Prediction and Metrics”, Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM '09), IEEE Computer Society, Washington, DC, USA, pp. 367-377.

163. [Richard 1999] Richard L. “Concepts and Application of Inferential Statistics”, http://faculty.vassar.edu/lowry/webtext.html, 1999.

164. [Richardson and Clarke 1981] Richardson D. and Clarke L. “A partition analysis method to increase program reliability”, Proceedings of the 5th international Conference on Software Engineering (San Diego, California, United States, March 09 - 12), International Conference on Software Engineering. IEEE Press, Piscataway, NJ, pp. 244-253, 1981.

165. [Rocacher 1988] Rocacher D., “Metrics definition for smalltalk”, Technical report, European Union ESPRIT Research Report 1257, 1988.

166. [Rombach 1987] Rombach D. “A Controlled Experiment on the Impact of Software Structure on Maintainability”, IEEE Transactions of Software Engineering, vol. SE-13, no. 3, pp. 344-354, 1987.

167. [Roper 1994] Roper M., "Software testing", London: McGraw-Hill Book Co., 1994.

168. [Rosenberg and Hyatt 1997] Rosenberg L. and Hyatt L. “Software Quality Metrics for Object-oriented Environments”, Crosstalk Journal, vol. 10, no. 4, Software Technology Support Center, Hill, UT.

169. [Schneberger 1997] Schneberger S. “Distributed computing environments: effects on software maintenance difficulty”, Journal of Systems and Software, vol. 37, no. 2, pp. 101 – 116, 1997.

170. [Sharble et al. 1993] Sharble R. and Samuel C. “The Object-Oriented Brewery: A Comparison of Two Object-Oriented Development Methods”, Software Engineering Notes, vol. 18, no. 2, pp 60 -73, 1993.

171. [Sheetz et al. 1992] Sheetz S., Tegarden D. and Monarchi D. “Measuring object oriented system complexity”, Working Paper, Univ. of Colorado, 1992.

172. [Shibata et al. 2007] Shibata K., Rinsaka K., Dohi T. and Okamura H. “Quantifying Software Maintainability Based on a Fault-Detection/Correction Model”, Proceedings of the PRDC 2007, pp. 35 – 42, 2007.

173. [Shooman 1983] Shooman M. “Software Engineering: Reliability, Development, and Management”, McGraw-Hill Inc., New York, NY, USA, 1983.

174. [Silva and Someren 2010] Silva L. and Someren M. “Evolutionary testing of object-oriented software”, Proceedings of the 2010 ACM Symposium on Applied Computing (SAC '10). ACM, New York, NY, USA, pp. 1126-1130, 2010.

175. [Sinha and Harrold 1999] Sinha S. and Harrold M. “Criteria for testing exception-handling constructs in Java programs”, Proceedings of the International Conference on Software Maintenance, pp. 265-274, 1999.

176. [Smith and Robson 1992] Smith M. and Robson D. “A Framework for Testing Object-Oriented Programs”, Journal of Object-Oriented Programming, vol. 5, no. 3, pp. 45 – 53, June 1992.

177. [Stevens et al. 1974] Stevens W., Myers G. and Constantine L. “Structured design”, IBM Systems Journal, vol. 13, no. 2, pp. 115–139, 1974.

178. [Sthamer et al. 2002] Sthamer H., Wegener J. and Baresel A. “Using evolutionary testing to improve efficiency and quality in software testing”, Proceedings of the 2nd Asia-Pacific Conference on Software Testing Analysis and Review (AsiaSTAR), pp. 22-24, 2002.

179. [Sultan et al. 2010] Sultan H., Ahmed S. and Mohammed E. “The limitations of genetic algorithms in software testing”, Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications - AICCSA 2010 (AICCSA '10), IEEE Computer Society, Washington, DC, USA, pp. 1-7, 2010.

180. [Tang et al. 1999] Tang M., Kao M. and Chen M. “An Empirical Study on Object Oriented Metrics,” Proceedings of Sixth International Software Metrics Symposium, pp. 242-249, 1999.

181. [Tegarden et al. 1992] Tegarden D., Sheetz S. and Monarchi D. “Effectiveness of traditional software metrics for object oriented systems”, Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, vol. 4, pp. 359 – 368, 1992.

182. [Tonella 2004] Tonella P. “Evolutionary testing of classes” SIGSOFT Software Engineering Notes, vol. 29, no. 4, pp. 119-128, 2004.

183. [Tracey et al. 1998] Tracey N., Clark J. and Mander K. “Automated program flaw finding using simulated annealing”, Software Engineering Notes, vol. 23, no. 2, pp. 73–81, 1998.

184. [Tsai et al. 1986] Tsai T., Lopez A., Rodreguez V., Volovik D. “An Approach to Measuring Data Structure Complexity”, COMPSAC86, pp. 240-246, 1986.

185. [Tse and Xu 1996] Tse T. and Xu Z. “Test case generation for class-level object-oriented testing”, Proceedings of the 9th International Software Quality Week, San Francisco, CA, May 1996. Software Research Inc., San Francisco, CA, pp. 4T4.0–4T4.12, 1996.

186. [VanderBrug and Minker 1975] VanderBrug G. and Minker J. “State-space problem-reduction, and theorem proving—some relationships”, Communications of the ACM, vol. 18, no. 2, pp. 107-119, 1975.

187. [Vessey and Weber 1984] Vessey and R. Weber, “Research on structured programming: An empiricist’s evaluation,” IEEE Trans. Software Eng., vol. SE-IO, pp. 394-407, 1984.

188. [Wappler and Lammermann 2005] Wappler S. and Lammermann F. “Using evolutionary algorithms for the unit testing of object-oriented software”, Proceedings of the 2005

conference on Genetic and evolutionary computation (GECCO '05), Hans-Georg Beyer (Ed.). ACM, NY, USA, pp. 1053-1060, 2005.

189. [Wappler and Wegener 2006a] Wappler S. and Wegener J. "Evolutionary unit testing of object-oriented software using strongly-typed genetic programming", Proceedings of the 8th annual conference on Genetic and evolutionary computation (GECCO '06). ACM, NY, USA, pp. 1925-1932, 2006.

190. [Wappler and Wegener, 2006b] Wappler S. and Wegener J. "Evolutionary unit testing of object-oriented software using a hybrid evolutionary algorithm", Proceedings of the IEEE World Congress on Computational Intelligence (WCCI-2006), Vancouver, Canada, IEEE Press, pp. 3193–3200, 2006.

191. [Watkins 1995] Watkins A. "The Automatic Generation of Software Test Data using Genetic Algorithms", Proceedings of the Fourth Software Quality Conference, pp. 300-309, July 1995.

192. [Weiser et al. 1985] Weiser M., Gannon J.and McMullin P., "Comparison of structural test coverage metrics", IEEE Software, vol. 2, no. 2, pp. 80-85, Mar. 1985.

193. [Weisfeld 2000] Weisfeld M. "The Object-Oriented Thought Process", SAMS Publishing, 2000.

194. [Welker and Oman 1997] Welker K. and Oman P. "Development and Application of an Automated Source Code Maintainability Index", Journal of Software Maintenance: Research and Practice, vol. 9, no. 3, pp. 127 – 159, 1997.

195. [Weyuker 1982] Weyuker E. "On testing non-testable programs", The Computer Journal, vol. 25, no. 4, pp. 465–470, November 1982.

196. [Weyuker 1988] Weyuker E. "Evaluating software complexity measures", IEEE Transactions on Software Engineering, vol. 14, pp. 1357-1365, 1988.

197. [Weyuker and Jeng 1991] Weyuker E. and Jeng B. "Analyzing Partition Testing Strategies", IEEE Transactions on Software Engineering, vol. 17, no. 7, pp. 703-711, Jul. 1991.

198. [Whitley 1993] Whitley L. "Cellular Genetic Algorithms", Proceedings of the 5th International Conference on Genetic Algorithms, Stephanie Forrest (Ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 658-665, 1993.

199. [Whitmire 1992] Whitmire S. "Measuring complexity in object-oriented software", Third International Conference Applications and Software Measurement, La Jolla, CA, 1992.

200. [Whittaker 2000] Whittaker J. "What Is Software Testing? And Why Is It So Hard?", IEEE Software, vol. 17, no. 1, pp. 70-79, 2000.

201. [Wilde and Huitt 1992] Wilde N. and Huitt R. "Maintenance support for object-oriented programs," IEEE Transactions on Software Engineering, vol. 18, no. 12, pp. 1038-1044, 1992.

202. [Xie et al. 2005] Xie T., Marinov D., Schulte W. and Notkin D. "Symstra: a framework for generating object-oriented unit tests using symbolic execution", Proceedings of the 11th

International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 365–381, 2005.

203. [Yang and Tempero 2007a] Yang H. and Tempero E. “Indirect Coupling As a Criteria for Modularity”, Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques (ACoM '07), IEEE Computer Society, Washington, DC, USA, pp. 10-11, 2007.
204. [Yang and Tempero 2007b] Yang H. and Tempero E. “Measuring the strength of indirect coupling”, Australian Software Engineering Conference, pp. 319-328, 2007.
205. [Yang et al. 2005] Yang H., Tempero E. and Berrigan R. “Detecting Indirect Coupling”, Australian Software Engineering Conference, pp. 212-221, 2005.
206. [Yang and Zhang 2009] Yang A. and Zhang W., “Based on Quantification Software Quality Assessment Method”, Journal of Software, Vol 4, No 10, pp. 1110-1118, 2009
207. [Yin 2002] Yin, R., “Case Study Research, Design and Methods”, 3rd ed. Newbury Park, Sage Publications, 2002.
208. [Yourdon and Constantine 1979] Yourdon E. and Constantine L. “Structured Design: Fundamentals of a Discipline of Computer Program and System Design”, Prentice-Hall, 1979.
209. [Zhou and Leung 2007] Zhou Y. and Leung H. “Predicting Object-Oriented Software Maintainability using Multivariate Adaptive Regression Splines”, Journal of Systems and Software, vol. 80, no. 8, pp. 1349 – 1361, 2007.
210. [Zhou and Xu 2008] Zhou Y. and Xu B. “Predicting the Maintainability of Open Source Software using Design Metrics”, Wuhan University Journal of Natural Sciences, vol. 13, no. 1, pp. 14 – 21, 2008.

APPENDIX A

Experimental Results for EasyMock and Hibernate

A1. Experiments with EasyMock

a). Following table tabulates LOC measure, IPC and Change metric of each class in different releases of EasyMock from versions 2.0 through version 2.5.

S.No.	Class Name	v2.0	v2.1	v2.3	v2.4	v2.5	IPC	Change
1	AbstractMatcher	25	25	25	25	27	5.12	2
2	AlwaysMatcher	4	4	4	4	5	0.58	1
3	And	14	14	14	14	16	3.44	2
4	Any	5	5	5	5	7	2.74	2
5	ArgumentsMatcher	3	3	3	3	3	0	0
6	ArrayEquals	29	29	29	29	30	5.1	1
7	ArrayMatcher	7	7	7	7	8	2.36	1
8	AssertionErrorWrapper	4	4	4	4	5	0.44	1
9	Contains	6	6	6	6	8	1.22	2
10	EasyMock	202	205	205	220	238	24.74	36
11	EndsWith	6	6	6	6	8	1.01	2
12	Equals	16	16	16	16	18	3.11	2
13	EqualsMatcher	2	2	2	2	3	0	1
14	EqualsWithDelta	9	9	9	9	11	1.98	2
15	ExpectedInvocation	43	43	43	42	44	10.4	3
16	ExpectedInvocationAndResult	7	7	7	7	9	1.18	2
17	ExpectedInvocationAndResults	8	8	8	8	10	0.27	2
18	Find	7	7	7	7	9	1.03	2
19	GreaterOrEqual	6	6	6	4	5	1.53	3
20	GreaterThan	6	6	6	4	5	1.05	3
21	InstanceOf	6	6	6	6	8	2.09	2
22	Invocation	35	38	38	50	57	10.57	22
23	IProxyFactory	3	3	3	3	3	0	0
24	JavaProxyFactory	4	4	4	4	4	0.28	0
25	LastControl	39	49	49	49	49	5.91	10
26	LegacyMatcherProvider	17	17	17	17	17	0.78	0
27	LessOrEqual	6	6	6	4	5	1.59	3
28	LessThan	6	6	6	4	5	0.25	3
29	Matches	6	6	6	6	8	2.98	2
30	MockControl	63	63	63	63	63	12.19	0
31	MockInvocationHandler	11	11	11	11	13	1.97	2
32	MocksBehavior	44	44	44	44	50	5.44	6
33	MocksControl	70	70	73	77	87	13.45	17
34	Not	8	8	8	8	10	1.24	2
35	NotNull	5	5	5	5	7	1.29	2
36	Null	5	5	5	5	7	1.03	2
37	ObjectMethodsFilter	21	21	21	25	37	6.45	16
38	Or	14	14	14	14	16	1.18	2
39	Range	18	18	18	18	20	3.6	2
40	RecordState	131	132	138	138	139	24.53	8
41	ReplayState	23	23	30	30	35	6.67	12
42	Result	15	18	10	10	17	8.14	18
43	Results	29	29	29	29	31	7.26	2
44	RuntimeExceptionWrapper	4	4	4	4	5	1.27	1
45	Same	12	12	12	12	14	2.1	2
46	StartsWith	6	6	6	6	8	2.56	2

S.No.	Class Name	v2.0	v2.1	v2.3	v2.4	v2.5	IPC	Change
47	ThrowableWrapper	4	4	4	4	5	1.63	1
48	UnorderedBehavior	30	30	30	30	32	5.63	2

A2. Experiments with Apache Tiles

Following table tabulates LOC measure, IPC and Change metric of each class in different releases of Apache Tiles from versions 2.0.3 through version 2.2.2.

S.No.	Class Name	v2.0.3	v2.0.4	v2.0.5	v2.0.6	v2.0.7	v2.1.0	v2.1.1	v2.1.2	v2.1.3	v2.1.4	v2.2.0	v2.2.1	v2.2.2	IPC	Change
1	AddAttributeTag	48	47	47	47	47	67	67	66	66	66	46	46	49	2.66	45
2	AddListAttributeTag	36	37	37	37	33	34	30	30	30	30	31	31	31	2.85	11
3	Attribute	94	101	101	134	134	146	146	183	187	187	225	225	237	2.65	143
4	BasicAttributeContext	98	106	106	109	109	56	56	56	56	56	56	56	56	2.69	64
5	BasicPreparerFactory	31	31	31	31	31	29	29	29	29	29	29	29	29	0.96	2
6	BasicPreparerFactoryTest	21	19	19	19	19	18	18	18	18	18	18	18	18	0.69	3
7	BasicTilesContainer	311	308	313	351	351	353	357	321	324	324	330	330	330	1.01	97
8	BasicTilesContainerTest	42	68	103	103	103	102	105	106	106	106	102	102	102	2.27	70
9	CachingTilesContainer	40	39	39	39	39	35	33	33	33	33	33	33	33	0.61	7
10	ChainedTilesContextFactory	73	81	85	85	85	89	97	97	91	91	91	91	91	1.73	30
11	ClassUtil	30	32	32	32	17	36	15	15	15	15	15	15	15	2.65	57
12	DefaultLocaleResolver	23	23	23	23	23	21	21	21	21	21	21	21	21	1.39	2
13	DefinitionManager	134	122	122	119	119	132	132	132	132	132	131	131	131	1.28	29
14	DefinitionsFactoryUtil	25	25	25	25	25	28	28	24	24	24	24	24	24	1.24	7
15	DefinitionTag	97	99	99	98	99	123	124	126	126	126	61	61	61	2.34	96
16	DigesterDefinitionsReader	117	136	151	159	159	202	202	221	221	221	226	226	226	1.56	109
17	ImportAttributeTag	39	38	38	38	43	60	65	65	65	68	49	49	49	2.18	50
18	InitContainerTag	194	193	193	193	194	201	211	211	211	211	206	206	206	1.08	24
19	InsertAttributeTag	57	57	57	57	58	64	66	109	109	109	87	87	90	1	77
20	JspTilesContextFactory	37	36	36	36	36	32	19	19	19	19	19	19	19	2.89	18
21	JspTilesRequestContext	41	42	42	38	38	38	45	68	68	68	68	68	68	2.31	35
22	JspUtil	44	45	45	19	19	53	53	85	85	85	143	143	143	3.35	151
23	JspWriterResponse	21	20	20	20	20	20	20	20	20	20	20	20	20	1.4	1
24	ListAttribute	28	28	28	28	28	11	11	11	11	11	11	11	11	1.29	17
25	MapEntry	42	42	42	42	42	42	42	42	42	42	42	42	42	0	0
26	MockDefinitionsReader	20	20	20	20	20	19	19	19	19	19	19	19	19	1.14	1
27	MockOnlyLocaleTilesContext	45	45	45	45	45	45	49	64	64	64	69	69	69	1.63	24
28	PutAttributeTag	55	54	54	54	55	39	40	34	34	34	61	61	61	2.68	52
29	PutListAttributeTag	37	38	38	38	38	53	53	53	53	53	53	53	56	1.61	19
30	RollingVectorEnumeration	24	24	24	24	24	25	25	25	25	25	25	25	25	1.97	1
31	SimpleMenuItem	52	52	52	52	52	52	52	52	52	52	52	52	52	0	0
32	TilesAccess	68	70	70	70	66	72	72	80	80	80	81	81	81	2.58	21
33	TilesAccessTest	26	26	26	26	26	39	39	33	33	33	33	33	33	1.13	19
34	TilesContainerFactory	161	161	161	161	161	187	277	277	277	277	286	286	286	4.63	125
35	UseAttributeTag	41	40	40	40	40	39	39	39	39	39	78	78	78	2.07	41

A3. Experiments with Hibernate

Following table tabulates LOC measure, IPC and Change metric of each class in different releases of Hibernate from versions 3.0a through version 3.2.5.

S.No.	Class Name	v3.0a	v3.0b	v3.0	v3.0.1	v3.0.2	v3.0.3	v3.1.0a	v3.1.0b	v3.1.0	v3.2.0	v3.2.1	v3.2.2	v3.2.3	v3.2.4	v3.2.5	IPC	Change
1	AbstractBatcher	139	149	159	160	161	161	164	164	186	191	191	191	191	191	191	20.46	52
2	AbstractComponentType	14	15	18	18	18	18	18	18	19	19	19	19	19	19	19	30.19	5
3	AbstractEvent	5	5	6	6	6	6	5	5	5	5	5	5	5	5	5	10.75	2
4	AbstractLazyInitializer	34	29	29	29	30	30	33	33	39	39	41	41	41	41	41	28.63	17
5	AbstractLockUpgradeEventListener	20	20	26	26	26	26	26	26	26	28	28	28	28	28	28	9.58	8
6	AbstractPropertyMapping	70	72	79	79	79	79	79	79	81	81	81	85	85	85	85	10.91	15
7	AbstractReassociateEventListener	23	23	25	25	25	25	27	27	27	27	27	26	26	26	26	8.82	5
8	AbstractSaveEventListener	80	94	96	96	96	96	107	112	113	122	122	122	122	122	122	26.54	42
9	AbstractType	23	27	39	39	39	39	46	46	46	46	46	46	46	46	46	9.84	23
10	AbstractVisitor	27	28	34	34	34	34	34	30	30	30	30	30	30	30	30	11.42	11
11	AnyType	87	91	95	96	96	97	98	105	109	109	109	109	109	109	109	6.19	22
12	ArrayType	45	46	48	48	48	48	47	48	49	49	49	49	49	49	49	14.49	6
13	Assigned	9	9	15	15	15	15	14	14	14	14	14	14	14	14	14	9.13	7
14	AssociationType	12	12	12	13	13	14	14	14	14	14	14	14	14	14	14	5.95	2
15	ASTPrinter	57	76	78	78	78	78	91	92	92	92	92	92	92	92	92	9.08	35
16	AutoFlushEvent	7	7	8	8	8	8	7	7	10	10	10	10	10	10	10	10.56	5
17	BagType	10	10	19	19	19	19	19	19	19	19	19	19	19	19	19	13.5	9
18	BasicCollectionPersister	73	76	81	81	102	102	97	97	97	97	103	103	103	103	103	17.14	40
19	BasicLazyInitializer	41	42	46	46	46	46	46	46	42	43	43	43	43	43	43	16.77	10
20	BasicPropertyAccessor	94	98	101	101	101	101	101	102	102	102	102	102	102	102	102	10.88	8
21	Batcher	24	25	28	28	28	28	30	30	32	33	33	33	33	33	33	21.3	9
22	BatcherFactory	3	3	2	2	2	2	2	2	3	3	3	3	3	3	3	4.19	2
23	BatchingBatcher	32	29	32	34	34	34	34	34	35	35	26	26	26	26	26	9.36	18
24	BatchingBatcherFactory	3	3	2	2	2	2	2	2	3	3	3	3	3	3	3	4.28	2
25	BatchingCollectionInitializer	45	44	40	40	40	40	40	40	40	40	40	40	40	40	40	16.81	5
26	BatchingEntityLoader	48	48	44	44	44	44	44	44	44	44	44	44	44	44	44	10.94	4
27	BigDecimalType	15	16	17	17	17	17	17	17	17	17	17	17	17	17	17	10.71	2
28	BlobImpl	28	33	33	33	33	33	33	33	33	33	33	33	33	33	33	12.01	5
29	BlobType	22	23	43	43	43	43	43	45	46	46	46	46	46	46	46	7.97	24
30	BooleanType	16	18	18	18	18	18	18	18	17	17	17	17	17	17	17	17.97	3
31	BytesHelper	26	6	6	6	6	6	6	6	6	6	6	6	6	6	6	5.42	20
32	ByteType	17	22	22	22	22	22	22	22	24	24	24	24	24	24	24	10.82	7
33	C3P0ConnectionProvider	58	60	60	60	61	61	61	61	61	61	61	83	83	83	83	1.57	25
34	Cache	17	17	20	20	21	21	21	21	21	21	21	21	21	21	21	6.38	4
35	CacheConcurrencyStrategy	18	19	17	17	17	17	17	17	17	19	19	19	19	19	19	2.42	5
36	CacheEntry	28	29	32	32	32	32	33	33	41	41	41	41	41	41	41	1.53	13
37	CacheProvider	4	6	7	7	7	7	7	7	7	7	7	7	7	7	7	10.6	3

S.No.	Class Name	v3.0a	v3.0b	v3.0	v3.0.1	v3.0.2	v3.0.3	v3.1.0a	v3.1.0b	v3.1.0	v3.2.0	v3.2.1	v3.2.2	v3.2.3	v3.2.4	v3.2.5	IPC	Change
38	CacheSynchronization	11	26	31	32	32	32	32	32	37	37	37	37	37	37	37	7.36	26
39	CalendarDateType	30	38	40	40	40	40	40	40	40	40	40	40	40	40	40	13.59	10
40	CalendarType	35	49	51	51	51	51	51	51	52	52	52	52	52	52	52	9.05	17
41	CGLIBLazyInitializer	36	36	42	42	42	42	50	50	59	61	61	61	64	64	64	10.48	28
42	CGLIBProxyFactory	21	21	24	24	24	24	23	23	23	25	25	25	25	25	25	4.13	6
43	CharacterType	18	20	20	20	20	20	20	20	21	21	21	21	21	21	21	11.16	3
44	ClassMetadata	28	28	31	31	31	31	35	36	36	36	36	36	36	36	36	13.83	8
45	ClassType	20	18	18	18	18	18	18	18	18	18	18	18	18	18	18	10.42	2
46	ClobImpl	34	39	41	41	41	41	41	41	41	41	41	41	41	41	41	14.59	7
47	ClobType	22	23	47	47	47	47	47	49	50	50	50	50	50	50	50	10.87	28
48	CollectionAction	30	34	38	38	38	38	42	42	42	45	45	45	45	45	45	18.55	15
49	CollectionEntry	107	108	100	100	100	100	91	91	91	102	102	102	102	102	102	23.71	29
50	CollectionKey	17	20	28	28	28	28	28	28	28	37	37	37	37	37	37	9.98	20
51	CollectionPersister	50	52	57	57	61	60	65	65	70	70	71	71	71	71	71	12.35	23
52	CollectionProperties	13	13	21	21	21	21	21	21	21	21	21	21	21	21	21	7.27	8
53	CollectionPropertyMapping	49	49	41	41	41	41	41	41	41	41	41	41	41	41	41	7.74	8
54	CollectionRemoveAction	12	12	12	12	12	12	15	15	15	15	15	15	15	15	15	21.42	3
55	CollectionStatistics	11	11	13	13	13	13	13	13	13	13	13	13	13	13	13	13.34	2
56	CollectionUpdateAction	25	25	25	25	25	25	28	28	28	28	28	28	28	28	28	14.95	3
57	Column	57	68	77	77	78	78	81	81	102	102	103	103	103	103	103	7.27	46
58	ColumnMetadata	19	19	19	19	19	19	19	19	23	23	23	23	23	23	23	1.79	4
59	Component	77	89	99	99	99	99	99	111	111	114	102	102	102	102	102	6.79	49
60	CompositeCustomType	77	77	81	81	81	81	80	89	91	91	91	91	91	91	91	15.32	16
61	CompositeUserType	22	24	24	24	23	23	23	23	23	23	23	23	23	23	23	4.77	3
62	ConfigHelper	32	32	32	32	32	32	32	32	42	42	52	52	52	52	52	20.46	20
63	Configurable	8	4	4	4	4	4	4	4	4	4	4	4	4	4	4	17.15	4
64	ConnectionProviderFactory	42	42	42	42	42	42	42	61	61	61	61	61	61	61	61	9.09	19
65	Constraint	25	27	28	28	28	28	28	28	28	28	32	32	32	32	32	9.78	7
66	Criteria	39	39	36	36	36	35	35	35	35	39	39	39	39	39	39	31.68	8
67	CriteriaImpl	218	218	199	199	199	199	205	203	203	209	209	209	209	209	209	11.31	33
68	CurrencyType	41	39	39	39	39	39	39	39	40	40	40	40	40	40	40	1.43	3
69	CustomType	43	44	59	59	59	59	59	61	63	63	67	67	67	67	67	15.94	24
70	DatabaseMetadata	46	51	51	51	51	51	51	51	54	58	70	70	70	70	70	15.99	24
71	DatasourceConnectionProvider	28	28	28	28	29	29	29	31	31	31	31	31	31	31	31	6.38	3
72	DateType	34	36	37	37	37	37	41	41	42	42	42	42	42	42	42	8.52	8
73	DefaultAutoFlushEventListener	23	18	22	22	22	22	22	22	21	21	21	21	21	21	21	2.26	10
74	DefaultDirtyCheckEventListener	17	12	14	14	14	14	14	14	13	13	13	13	13	13	13	2.04	8
75	DefaultEvictEventListener	30	31	36	36	36	36	37	37	38	38	38	38	38	38	38	8.02	8
76	DefaultInitializeCollectionEventListener	32	31	44	44	44	44	45	45	45	45	45	45	45	45	45	7.19	15
77	DefaultLockEventListener	26	28	27	27	27	27	29	29	29	29	29	29	29	29	29	11.11	5
78	DefaultNamingStrategy	9	9	9	9	9	9	9	9	18	18	18	18	18	18	18	3.31	9
79	DefaultRefreshEventListener	42	45	50	50	50	50	51	57	61	61	61	61	61	61	61	14.04	19
80	DefaultReplicateEventListener	36	48	51	51	51	51	55	55	55	55	55	56	56	56	56	15.84	20
81	DefaultSaveEventListener	24	13	15	15	15	15	15	15	15	15	15	15	15	15	15	11.21	13

S.No.	Class Name	v3.0a	v3.0b	v3.0	v3.0.1	v3.0.2	v3.0.3	v3.1.0a	v3.1.0b	v3.1.0	v3.2.0	v3.2.1	v3.2.2	v3.2.3	v3.2.4	v3.2.5	IPC	Change
82	DefaultUpdateEventListener	49	15	17	17	17	17	17	17	17	17	17	17	17	17	17	11.75	36
83	Delete	23	23	23	23	23	23	30	30	30	30	33	33	33	33	33	11.02	10
84	DeleteEvent	13	13	14	14	14	14	13	15	15	15	15	15	15	15	15	19.42	4
85	DeleteEventListener	4	4	4	4	4	4	4	4	4	4	6	6	6	6	6	16.59	2
86	DenormalizedTable	13	13	13	13	13	13	13	13	30	30	30	33	33	33	33	2.46	20
87	DependantValue	7	8	14	14	14	14	14	14	14	14	14	14	14	14	14	4.22	7
88	DetailedSemanticException	23	23	19	19	19	19	19	19	19	19	19	19	19	19	19	0.3	4
89	DirectPropertyAccessor	38	40	43	43	43	43	43	44	44	44	50	50	50	51	51	13.03	13
90	DirtyCheckEvent	2	2	3	3	3	3	2	2	5	5	5	5	5	5	5	17.53	5
91	DirtyCollectionSearchVisitor	16	19	20	20	20	20	21	21	21	21	21	21	21	21	21	13.42	5
92	DoubleType	13	15	15	15	15	15	15	15	16	16	16	16	16	16	16	12.1	3
93	DriverManagerConnectionProvider	74	71	71	71	72	72	72	72	72	72	72	72	72	72	72	8.42	4
94	DTDEntityResolver	26	26	27	27	27	27	27	27	28	37	36	36	36	36	36	28.47	12
95	EhCache	46	41	52	52	53	53	53	53	53	53	54	54	54	54	54	9.09	18
96	EhCacheProvider	6	22	23	23	23	23	23	25	25	25	36	36	36	36	36	0	30
97	EntityAction	29	33	33	33	33	33	33	33	34	35	35	35	35	36	36	30.32	7
98	EntityDeleteAction	32	37	47	47	47	47	45	45	64	66	67	67	67	67	67	11.35	39
99	EntityEntry	58	58	53	53	53	53	67	71	70	102	102	102	102	102	102	2.08	56
100	EntityIdentityInsertAction	22	27	27	27	27	27	27	27	48	58	59	59	59	59	59	16.57	37
101	EntityInsertAction	25	32	41	41	42	42	42	42	65	66	67	67	67	68	68	6.17	43
102	EntityKey	21	24	35	35	35	35	35	35	35	54	54	54	54	54	54	9.52	33
103	EntityLoader	37	41	49	49	49	46	42	42	21	21	21	21	21	21	21	8.64	40
104	EntityPersister	76	78	85	86	86	86	90	91	103	105	120	123	123	123	123	18.13	47
105	EntityStatistics	11	11	13	13	13	13	13	13	15	15	15	15	15	15	15	16.37	4
106	EntityUniqueKey	18	21	25	25	25	25	28	28	33	42	42	42	42	42	42	8.73	24
107	EntityUpdateAction	43	50	61	61	61	61	61	61	84	87	89	89	89	89	89	14.74	46
108	Environment	132	139	143	143	144	144	144	144	149	169	171	172	172	173	173	10.51	41
109	ErrorCounter	31	36	34	34	34	34	30	30	30	30	30	30	30	30	30	27.28	11
110	EvictEvent	6	6	7	7	7	7	6	6	6	6	6	6	6	6	6	11.12	2
111	EvictVisitor	6	22	23	23	23	23	23	23	23	23	23	23	23	23	23	30.51	17
112	Example	108	108	113	113	113	113	113	113	117	117	120	120	120	120	120	14.5	12
113	Expression	36	39	6	6	6	6	6	6	6	6	6	6	6	6	6	12.17	36
114	Fetchable	3	4	4	4	4	6	6	6	6	6	6	6	6	6	6	14.49	3
115	FetchMode	17	19	17	17	17	17	17	17	17	17	17	17	17	17	17	5.16	4
116	Filter	4	4	7	7	7	7	7	7	9	9	9	9	9	9	9	0.81	5
117	FilterDefinition	13	13	16	16	17	17	17	17	16	16	17	17	17	17	17	11.09	6
118	FilterImpl	23	23	33	33	35	35	39	37	38	38	38	38	38	38	38	4.39	19
119	FloatType	13	15	15	15	15	15	15	15	16	16	16	16	16	16	16	11.01	3
120	FlushEvent	2	2	3	3	3	3	2	2	2	2	2	2	2	2	2	10.66	2
121	FlushMode	20	20	20	20	20	20	20	20	20	21	24	24	24	24	24	8.99	4
122	FlushVisitor	15	13	15	15	15	15	15	15	15	15	15	15	15	15	15	17.88	4
123	ForeignGenerator	22	21	23	23	23	23	27	27	27	27	27	27	27	27	27	5.3	7
124	ForeignKey	38	40	40	40	40	40	55	55	55	55	55	55	55	55	55	2.11	17
125	ForeignKeyDirection	10	12	12	12	12	12	12	12	12	12	12	12	12	12	12	16.97	2

S.No.	Class Name	v3.0a	v3.0b	v3.0	v3.0.1	v3.0.2	v3.0.3	v3.1.0a	v3.1.0b	v3.1.0	v3.2.0	v3.2.1	v3.2.2	v3.2.3	v3.2.4	v3.2.5	IPC	Change
126	ForeignKeys	46	54	66	67	67	67	67	67	67	67	68	68	68	68	68	13.69	22
127	Formula	16	17	17	17	17	17	17	17	17	17	18	18	18	18	18	1.56	2
128	ForUpdateFragment	36	33	33	33	33	33	34	34	34	34	34	34	34	34	34	5.25	4
129	FromReferenceNode	23	28	32	32	32	32	32	33	33	33	33	33	33	33	33	2.6	10
130	FrontBaseDialect	23	23	23	23	23	23	23	23	25	32	32	32	32	32	32	0	9
131	GetGeneratedKeysHelper	27	27	27	27	27	27	27	27	25	28	28	28	28	28	28	7.99	5
132	Getter	7	8	10	10	10	10	10	11	11	11	11	11	11	11	11	20.16	4
133	GUIDGenerator	22	22	22	22	23	23	22	22	22	22	22	22	22	22	22	8.18	2
134	HashtableCache	16	16	20	20	21	21	21	21	21	21	21	21	21	21	21	4.14	5
135	HibernateProxyHelper	14	5	5	5	5	5	5	5	5	5	5	5	5	5	5	1.57	9
136	HibernateService	105	105	54	54	54	54	54	54	54	54	54	54	54	54	54	9.82	51
137	HibernateServiceMBean	60	60	64	64	64	64	64	64	64	64	64	66	66	66	66	0.84	6
138	HqlLexer	23	20	14	14	14	14	16	19	19	19	19	19	19	19	19	5.67	14
139	HqlParser	98	102	126	127	127	127	133	137	141	141	141	141	141	141	141	2	43
140	HSQLDialect	96	105	113	113	113	113	128	129	126	141	141	143	146	146	146	13.55	56
141	IdentifierBagType	10	10	13	13	13	13	13	13	13	13	13	13	13	13	13	5.16	3
142	IdentifierGenerator	6	5	6	6	6	6	6	6	6	6	6	6	6	6	6	2.82	2
143	IdentifierGeneratorFactory	44	42	49	49	52	52	54	54	54	55	55	55	55	55	55	20.76	15
144	IdentityGenerator	6	7	7	7	7	7	7	7	7	52	52	52	52	52	52	12.83	46
145	IdentityMap	67	80	80	80	80	80	81	87	87	87	87	87	87	87	87	8.66	20
146	ImmutableType	7	7	9	9	9	9	9	9	9	9	9	9	9	9	9	11.28	2
147	ImprovedNamingStrategy	14	14	14	14	14	14	14	14	23	23	23	23	23	23	23	28.82	9
148	IncrementGenerator	40	43	43	43	54	54	54	54	50	50	50	50	50	50	50	20.09	18
149	Index	30	34	36	36	36	36	36	36	36	36	36	36	36	36	36	29.26	6
150	IndexedCollection	19	19	29	29	29	29	29	29	29	29	29	29	29	29	29	12.55	10
151	IndexNode	33	46	57	57	57	57	57	59	59	59	59	59	59	59	59	6.29	26
152	InformixDialect	40	40	41	41	41	41	67	68	68	68	68	68	68	68	68	1.04	28
153	InFragment	36	36	35	35	35	35	36	36	36	36	36	36	36	36	36	22.64	2
154	InitializeCollectionEvent	6	6	7	7	7	7	6	6	6	6	6	6	6	6	6	2.63	2
155	Insert	40	40	44	44	44	44	44	44	44	45	45	45	45	45	45	2.71	5
156	InstantiationException	6	8	8	8	8	8	10	10	10	10	10	10	10	10	10	10.83	4
157	IntegerType	17	22	22	22	22	22	22	22	24	24	24	24	24	24	24	16.81	7
158	InterbaseDialect	31	31	31	31	31	32	32	34	40	40	40	40	40	40	40	10.11	9
159	Interceptor	15	15	18	18	18	18	18	18	22	22	22	22	22	22	22	3.31	7
160	IteratorImpl	66	66	67	67	63	63	65	65	65	65	65	65	65	65	67	15.63	9
161	JDBCExceptionReporter	14	29	29	29	33	33	33	33	33	33	34	34	34	34	34	21.03	20
162	JDBCTransaction	49	53	84	84	84	87	87	87	96	96	96	96	96	96	96	9.68	47
163	JDBCTransactionFactory	8	8	8	8	8	10	10	10	10	14	13	13	13	13	13	5.88	7
164	Join	51	58	59	59	59	59	59	59	59	59	69	69	69	69	69	8.58	18
165	JoinedSubclass	13	14	14	14	14	14	14	16	16	16	16	16	16	16	16	2.46	3
166	JoinedSubclassEntityPersister	260	241	240	240	240	240	252	252	252	252	262	262	263	263	263	3.53	43
167	JoinFragment	19	25	29	29	29	29	29	29	29	29	29	29	29	29	29	10.21	10
168	JoinHelper	22	27	28	28	28	28	28	28	28	28	28	28	28	28	28	5.17	6
169	JoinProcessor	54	42	56	56	56	56	56	58	60	60	60	60	60	60	60	5.59	30

S.No.	Class Name	v3.0a	v3.0b	v3.0	v3.0.1	v3.0.2	v3.0.3	v3.1.0a	v3.1.0b	v3.1.0	v3.2.0	v3.2.1	v3.2.2	v3.2.3	v3.2.4	v3.2.5	IPC	Change
170	JoinSequence	97	99	101	101	112	112	114	114	114	114	114	114	114	114	114	9.07	17
171	JTATransaction	82	90	96	97	97	104	104	104	113	114	114	114	114	114	114	18.94	32
172	JTATransactionFactory	28	28	25	25	25	27	27	27	27	37	38	38	38	38	38	5.79	16
173	KeyValue	8	9	9	9	10	10	10	10	10	10	10	10	10	10	10	4.9	2
174	LazyInitializationException	6	4	4	4	4	4	4	4	4	4	4	4	4	4	4	5.46	2
175	LazyInitializer	14	14	14	14	15	15	17	17	17	17	17	17	17	17	17	10.18	3
176	LinkedHashCollectionHelper	18	18	18	18	18	18	18	18	18	18	40	40	40	40	40	12.92	22
177	ListType	9	9	23	23	23	23	23	23	23	23	23	23	23	23	23	5.47	14
178	Loadable	17	17	17	17	17	17	20	20	20	20	20	20	20	20	20	14.38	3
179	LoadEvent	30	30	31	31	31	31	30	30	33	33	33	33	33	33	33	3.56	5
180	LoadEventListener	12	12	12	12	12	31	31	31	31	31	31	31	31	31	31	15.07	19
181	LocaleType	23	25	26	26	26	26	26	26	27	27	27	27	27	27	27	6.13	4
182	LockEvent	16	16	17	17	17	17	16	16	16	16	16	16	16	16	16	7.62	2
183	LockMode	23	23	23	23	23	23	23	23	23	26	26	26	26	26	26	10.95	3
184	LongType	17	22	22	22	22	22	22	22	24	24	24	24	24	24	24	27.1	7
185	ManyToOne	8	9	9	12	12	12	24	24	24	24	24	24	24	24	24	6.54	16
186	ManyToOneType	35	35	40	43	43	45	46	50	63	63	61	61	61	61	61	17.31	30
187	MapAccessor	18	18	21	21	21	21	21	21	21	21	21	21	21	21	21	19.75	3
188	MapProxy	20	21	21	21	21	21	21	21	21	24	24	24	24	24	24	8.86	4
189	MapProxyFactory	9	9	10	10	10	10	10	10	10	14	14	14	14	14	14	10.44	5
190	MapType	23	23	34	34	34	34	33	34	34	34	34	34	34	34	34	2.56	13
191	MatchMode	24	24	22	22	22	22	22	22	22	22	22	22	22	22	22	8	2
192	MckoiDialect	30	30	44	44	44	44	44	45	45	52	52	52	52	52	52	11.63	22
193	MessageHelper	37	37	84	84	84	84	84	84	84	90	90	90	90	90	90	24.14	53
194	MetaType	36	35	41	41	41	41	41	42	43	43	43	43	43	43	43	13.25	9
195	MethodNode	23	38	70	70	72	72	72	74	74	74	74	74	74	74	74	4.04	51
196	MutableType	6	6	10	10	10	10	11	11	11	11	11	11	11	11	11	17.25	5
197	MySQLDialect	141	151	161	160	160	160	164	166	168	169	170	174	174	174	174	11.58	35
198	NamedQueryDefinition	21	22	22	22	22	22	22	26	38	38	38	38	38	38	38	14.43	17
199	NamedQueryLoader	25	24	24	24	24	24	24	24	26	26	26	26	26	26	26	13.14	3
200	NamedSQLQueryDefinition	24	10	18	18	18	18	18	25	29	29	23	23	23	23	23	19.38	39
201	NameGenerator	2	14	14	14	14	14	14	14	14	14	14	14	14	14	14	17.42	12
202	NamingStrategy	6	6	6	6	6	6	6	6	11	11	11	11	11	11	11	9.84	5
203	NestableRuntimeException	29	29	29	29	29	29	29	29	36	36	36	36	36	36	36	4.4	7
204	NoArgSQLFunction	12	12	17	17	18	18	18	18	18	18	18	18	18	18	18	8.8	6
205	NonBatchingBatcher	8	8	9	9	9	9	9	9	10	10	9	9	9	9	9	7.85	3
206	NonBatchingBatcherFactory	3	3	2	2	2	2	2	2	3	3	3	3	3	3	3	2.9	2
207	NonstrictReadWriteCache	36	37	39	39	39	39	39	39	39	37	37	37	37	37	37	1.65	5
208	NullableType	36	36	43	43	43	43	43	45	49	61	61	64	64	64	64	8.89	28
209	OneToMany	27	28	33	37	37	37	37	37	37	37	37	37	37	37	37	16.89	10
210	OneToManyPersister	90	93	96	96	96	96	100	100	102	102	118	118	118	118	118	10.47	28
211	OneToOne	26	27	32	32	36	36	36	36	36	36	36	36	36	36	36	9.77	10
212	OneToOneType	26	26	26	25	37	37	37	38	39	39	39	39	39	39	39	13	15
213	OnLockVisitor	18	18	19	19	19	19	18	18	18	18	18	18	18	18	18	10.5	2

S.No.	Class Name	v3.0a	v3.0b	v3.0	v3.0.1	v3.0.2	v3.0.3	v3.1.0a	v3.1.0b	v3.1.0	v3.2.0	v3.2.1	v3.2.2	v3.2.3	v3.2.4	v3.2.5	IPC	Change
214	OnReplicateVisitor	17	17	19	19	19	21	21	21	21	21	21	20	20	20	20	8.91	5
215	OnUpdateVisitor	19	19	21	21	21	21	19	19	19	19	19	18	18	18	18	5.7	5
216	OracleDialect	14	20	33	33	33	33	33	25	29	29	29	29	29	29	34	18.2	36
217	OracleJoinFragment	43	43	45	45	45	45	44	44	44	44	44	44	44	44	44	12.02	3
218	Order	15	15	18	18	18	18	18	30	30	30	30	30	30	30	30	4.36	15
219	OSCache	25	25	29	29	30	30	30	30	30	30	30	30	30	30	30	7.92	5
220	OuterJoinableAssociation	46	46	47	47	58	57	57	57	57	57	57	57	57	57	57	4.62	13
221	OuterJoinLoadable	13	14	16	16	16	16	18	18	20	20	20	20	20	20	20	9.71	7
222	PathExpressionParser	215	215	210	210	210	210	210	210	210	210	210	210	210	210	210	10.82	5
223	PersistentIdentifierGenerator	10	10	10	10	14	14	14	14	14	14	14	14	14	14	14	7.85	4
224	PersisterFactory	39	39	44	44	44	44	44	44	44	44	44	44	44	44	44	41.57	5
225	PointbaseDialect	21	21	21	21	21	21	21	21	21	28	28	28	28	28	28	1.58	7
226	PostgreSQLDialect	90	91	97	97	102	104	112	113	135	135	136	137	138	138	138	21.78	48
227	PreprocessingParser	58	59	58	58	58	58	58	58	58	58	58	58	58	58	58	33.84	2
228	PropertiesHelper	25	25	25	25	25	25	25	25	55	55	54	54	54	54	54	1.29	31
229	Property	66	70	58	58	58	58	58	58	92	94	95	95	95	95	95	9.94	53
230	PropertyAccessorFactory	17	19	36	36	36	36	36	36	39	39	39	39	39	39	39	12.89	22
231	ProxoolConnectionProvider	71	71	65	65	66	66	66	66	66	66	66	66	66	66	66	18.85	7
232	ProxyFactory	8	8	9	9	9	9	9	9	9	14	14	14	14	14	14	13.22	6
233	ProxyVisitor	15	15	16	16	16	16	18	18	18	18	18	18	18	18	18	10.36	3
234	Query	78	80	82	86	86	85	85	85	85	87	89	89	89	89	89	3.45	13
235	Queryable	8	8	9	9	9	9	17	17	19	19	26	26	26	26	26	4.03	18
236	QueryableCollection	11	12	14	14	14	14	14	14	15	15	16	16	16	16	16	3.29	5
237	QueryCache	10	10	12	12	12	12	12	12	12	12	12	12	12	12	12	8.57	2
238	QueryImpl	43	43	42	42	42	42	42	42	43	43	43	43	43	43	43	13.43	2
239	QueryJoinFragment	35	37	37	37	37	37	37	37	37	37	37	37	37	37	37	11.02	2
240	QueryKey	59	64	67	67	69	69	69	69	69	76	76	76	76	76	76	14.92	17
241	QueryNode	46	29	43	43	43	43	49	40	40	40	40	40	40	40	40	3.2	46
242	QuerySplitter	48	49	49	49	49	55	55	55	57	57	57	57	57	57	57	17.4	9
243	QueryStatistics	19	19	24	24	24	24	24	24	24	25	25	25	25	25	25	18.97	6
244	QueryTranslator	22	23	28	29	29	29	32	32	35	36	36	36	36	36	36	22.93	14
245	QueryTranslatorFactory	29	5	5	5	5	5	5	5	5	5	5	5	5	5	5	4.63	24
246	QueryTranslatorImpl	461	454	457	458	460	460	465	465	463	467	467	469	469	469	469	3.45	26
247	ReadOnlyCache	31	32	34	34	34	34	34	34	34	34	34	34	34	34	34	0.17	3
248	ReadWriteCache	108	117	123	123	123	123	123	123	123	123	123	123	123	123	123	6.97	15
249	ReattachVisitor	18	18	28	28	28	28	28	24	24	24	24	27	27	27	27	15.32	17
250	ReflectHelper	92	93	97	97	97	97	98	98	102	80	78	78	78	78	78	8.58	34
251	RefreshEvent	12	12	13	13	13	13	12	12	12	12	12	12	12	12	12	12.11	2
252	RefreshEventListener	4	4	4	4	4	4	4	6	6	6	6	6	6	6	6	11.89	2
253	ReplicateEvent	18	18	19	19	19	19	18	18	18	18	18	18	18	18	18	5.46	2
254	ReplicationMode	26	26	24	24	24	24	24	24	24	24	24	24	24	24	24	6.46	2
255	RootClass	71	72	73	73	80	80	89	89	93	93	93	93	93	93	93	14.33	22
256	SchemaExportTask	90	90	90	90	90	90	90	90	89	89	89	89	89	91	91	22.85	3
257	SchemaUpdate	82	85	85	85	85	85	85	90	91	84	84	84	84	84	84	9.48	16

S.No.	Class Name	v3.0a	v3.0b	v3.0	v3.0.1	v3.0.2	v3.0.3	v3.1.0a	v3.1.0b	v3.1.0	v3.2.0	v3.2.1	v3.2.2	v3.2.3	v3.2.4	v3.2.5	IPC	Change
258	SchemaUpdateTask	78	78	77	77	77	77	77	77	78	78	78	78	78	80	80	12.46	4
259	ScrollableResults	43	43	43	45	45	45	45	45	45	45	45	45	45	45	45	17.2	2
260	SecondLevelCacheStatistics	16	16	27	27	27	27	27	27	27	27	27	27	27	27	27	1.89	11
261	Select	41	54	59	59	59	59	58	58	58	58	58	58	58	58	58	2.66	19
262	SelectFragment	56	61	63	63	63	63	63	63	63	63	63	63	63	63	63	14.15	7
263	SequenceGenerator	46	45	45	45	45	45	45	45	44	45	45	45	45	45	45	13.23	3
264	SequenceHiLoGenerator	27	26	26	26	26	26	26	26	26	26	29	29	29	29	29	4.14	4
265	SerializableProxy	31	24	27	27	27	27	27	27	27	27	27	27	27	27	27	3.58	10
266	SerializableType	30	29	30	30	30	30	30	30	30	30	30	30	30	30	30	6.65	2
267	SerializationHelper	32	49	49	49	49	49	49	49	49	51	51	51	51	51	51	13.3	19
268	Session	66	70	75	75	75	75	78	79	77	77	77	78	78	79	79	11.84	17
269	SessionFactory	30	32	33	37	37	38	40	40	41	41	41	41	41	41	41	5.49	11
270	SessionFactoryHelper	41	83	79	79	80	80	80	86	87	88	88	88	88	88	88	16.72	55
271	SessionFactoryImplementor	40	46	37	38	38	38	39	42	49	49	53	53	53	53	53	35.2	31
272	SessionFactoryStub	61	61	59	60	60	61	64	64	68	68	68	68	68	68	68	7.52	11
273	Setter	6	7	8	8	8	8	8	8	8	8	8	8	8	8	8	11.2	2
274	Settings	93	113	123	123	126	127	127	127	133	137	145	148	148	151	151	32.76	58
275	SetType	9	9	17	17	17	17	17	17	17	17	17	17	17	17	17	24.99	8
276	ShortType	17	22	22	22	22	22	22	22	24	24	24	24	24	24	24	30.57	7
277	SimpleExpression	26	26	28	28	28	28	28	39	40	40	40	40	40	40	40	16.28	14
278	SimpleSelect	85	85	85	85	85	85	84	84	85	85	85	85	85	85	85	10.65	2
279	SimpleValue	75	83	93	93	103	103	103	103	104	104	104	104	104	104	104	12.3	29
280	SortedMapType	14	14	21	21	23	23	23	23	23	23	23	23	23	23	23	5.94	9
281	SortedSetType	14	14	20	20	22	22	22	22	22	22	22	22	22	22	22	13.9	8
282	SQLFunction	7	7	9	9	10	10	10	10	10	10	11	11	11	11	11	7.1	4
283	SqlGenerator	15	21	47	47	50	50	50	53	72	72	74	74	74	74	74	14.59	59
284	SqlNode	8	10	9	9	9	9	9	9	9	9	9	9	9	9	9	16.97	3
285	StandardQueryCache	48	51	54	54	54	54	54	54	65	65	67	67	67	67	67	9.22	19
286	StandardSQLFunction	9	9	21	21	21	21	21	21	21	21	25	25	25	25	25	18.21	16
287	Statistics	34	34	38	38	40	40	40	40	42	42	42	42	42	42	42	5.1	8
288	StatisticsImpl	177	177	204	204	212	214	214	214	225	226	226	226	226	226	226	22.05	49
289	StatisticsImplementor	20	20	23	23	25	25	25	25	26	26	26	26	26	26	26	15.88	6
290	StatisticsService	79	79	83	83	85	85	85	85	87	87	87	87	87	87	87	26.26	8
291	Status	17	19	19	19	19	19	20	20	20	21	21	21	21	21	21	9.08	4
292	StringHelper	108	111	135	135	135	135	136	149	150	150	156	156	156	156	156	4.39	48
293	StringType	16	14	14	14	14	14	14	14	15	15	15	15	15	15	15	5.81	3
294	Subclass	62	63	63	63	64	64	65	69	72	72	79	79	79	79	79	23.22	17
295	SwarmCache	23	23	27	27	28	28	28	28	28	28	28	28	28	28	28	9.63	5
296	SwarmCacheProvider	9	15	16	16	16	16	16	16	16	16	16	16	16	16	16	8.7	7
297	SybaseDialect	72	72	87	86	87	88	89	89	100	100	100	120	122	122	122	2.33	52
298	TableHiLoGenerator	26	25	27	27	27	27	27	27	27	27	27	27	28	28	28	7.28	4
299	TableMetadata	65	68	68	68	68	68	68	68	68	68	63	63	63	63	63	9.14	8
300	TimestampType	46	50	51	51	54	52	52	52	54	54	54	54	54	54	54	11.3	12
301	TimeType	32	38	39	39	39	39	44	44	45	45	45	45	45	45	45	21.69	13

S.No.	Class Name	v3.0a	v3.0b	v3.0	v3.0.1	v3.0.2	v3.0.3	v3.1.0a	v3.1.0b	v3.1.0	v3.2.0	v3.2.1	v3.2.2	v3.2.3	v3.2.4	v3.2.5	IPC	Change
302	TimeZoneType	18	17	18	18	18	18	18	18	19	19	19	19	19	19	19	22.56	3
303	ToOne	18	20	23	26	26	29	32	32	32	32	32	32	32	32	32	3.01	14
304	Transaction	5	5	7	7	7	8	8	8	10	13	13	13	13	13	13	22.18	8
305	TransactionalCache	32	33	35	35	35	35	35	35	35	38	38	38	38	38	38	10.86	6
306	TransactionFactory	7	7	15	15	15	17	17	17	19	22	22	22	22	22	22	4.62	15
307	TreeCache	40	24	45	45	60	60	60	63	63	66	66	66	66	66	66	0.96	58
308	TreeCacheProvider	4	24	27	27	27	27	27	27	27	36	38	38	38	38	38	9.89	34
309	TwoPhaseLoad	42	47	51	51	51	51	52	52	63	64	64	64	64	64	64	5.71	22
310	Type	34	39	46	46	46	46	48	49	51	51	51	51	51	51	51	12.99	17
311	TypedValue	20	21	22	22	22	22	22	22	22	22	22	22	22	22	22	13.13	2
312	UnionSubclass	12	15	15	15	16	16	16	16	16	16	16	16	16	16	16	14.05	4
313	UniqueKey	14	14	19	19	19	19	19	19	19	19	29	29	29	29	29	18.85	15
314	UniqueKeyLoadable	8	7	5	5	5	5	5	5	5	5	5	5	5	5	5	11.19	3
315	Update	60	60	64	64	64	64	81	81	84	84	84	84	84	84	84	28.16	24
316	UpdateTimestampsCache	33	36	42	42	42	42	42	42	42	42	42	42	42	42	42	9.15	9
317	UserSuppliedConnectionProvider	9	9	8	8	9	9	9	9	9	9	9	9	9	9	9	3.02	2
318	UserType	16	18	18	18	18	18	18	18	18	18	18	18	18	18	18	14.16	2
319	Value	18	20	22	22	22	22	22	22	22	22	22	22	22	22	22	4.11	4
320	Versioning	26	31	31	31	31	31	31	31	31	31	31	32	32	32	32	5.62	6
321	VersionType	3	5	6	6	6	6	6	6	7	7	7	7	7	7	7	4.39	4
322	WebSphereTransactionManagerLookup	21	21	21	21	21	21	21	21	26	26	26	26	26	26	26	2.96	5
323	WhereParser	217	217	227	227	227	227	227	227	227	227	227	227	227	227	227	6.09	10
324	WrapVisitor	52	51	54	54	54	54	55	55	55	55	55	55	55	55	55	13.02	5
325	XMLHelper	29	29	32	32	32	32	32	32	40	40	40	40	40	40	40	7.5	11

A4. Experiments with Apache Velocity

Following table tabulates LOC measure, IPC and Change metric of each class in different releases of Apache Velocity from versions 1.0 through version 1.7.

S.No.	Class Name	v1.0	v1.0.1	v1.1	v1.2	v1.3	v1.3.1	v1.4	v1.5	v1.6	v1.6.1	v1.6.2	v1.6.3	v1.6.4	v1.7	Change	IPC
1	AbstractContext	77	77	81	74	74	74	74	71	72	72	72	72	72	72	15	0.49
2	AnakiaTask	278	278	278	291	291	291	290	384	384	384	384	384	384	384	108	2.53
3	ArrayIterator	30	30	30	30	36	36	36	36	36	36	36	36	36	36	6	0.38
4	ASTAddNode	45	45	45	46	46	46	46	60	34	34	34	34	34	34	41	1.06
5	ASTAndNode	31	42	47	46	46	46	46	47	42	42	42	42	42	42	23	0.58
6	ASTBlock	29	29	29	32	32	32	32	33	32	32	32	32	32	32	5	0.19
7	ASTDirective	73	73	74	76	76	76	75	117	118	118	118	118	118	142	71	0.9
8	ASTDivNode	53	53	53	54	54	54	54	63	34	34	34	34	34	34	39	0.57
9	ASTElseIfStatement	31	31	31	34	34	34	34	35	34	34	34	34	34	34	5	0.47
10	ASTElseStatement	24	24	24	24	24	24	22	23	22	22	22	22	22	22	4	0.29
11	ASTEQNode	51	51	51	52	52	52	58	82	82	82	82	82	82	82	31	0.83

S.No.	Class Name	v1.0	v1.0.1	v1.1	v1.2	v1.3	v1.3.1	v1.4	v1.5	v1.6	v1.6.1	v1.6.2	v1.6.3	v1.6.4	v1.7	Change	IPC
12	ASTEscape	33	33	33	33	33	33	34	37	36	36	36	36	36	33	8	0.38
13	ASTEscapedDirective	26	26	26	26	26	26	26	30	29	29	29	29	29	26	8	0.26
14	ASTExpression	29	29	29	29	29	29	29	30	29	29	29	29	29	29	2	0.5
15	ASTFalse	26	26	27	27	27	27	27	28	27	27	27	27	27	27	3	0.11
16	ASTGENode	28	47	47	48	48	48	54	62	59	59	70	70	70	70	48	1.07
17	ASTGTNode	28	47	47	48	48	48	54	62	59	59	70	70	70	70	48	0.52
18	ASTIdentifier	77	77	92	92	96	96	117	122	136	136	136	136	136	136	59	0.46
19	ASTIfStatement	43	43	43	46	46	46	46	47	46	46	46	46	46	46	5	0.18
20	ASTIntegerRange	56	56	56	55	55	55	55	59	57	57	57	57	57	57	7	0.23
21	ASTLENode	28	47	47	48	48	48	54	62	59	59	70	70	70	70	48	0.87
22	ASTLTNode	28	47	47	48	48	48	54	62	59	59	70	70	70	70	48	0.63
23	ASTMethod	102	102	129	131	131	131	121	199	227	227	227	227	227	181	191	3.89
24	ASTModNode	51	51	51	52	52	52	52	63	34	34	34	34	34	34	41	1.12
25	ASTMulNode	44	44	44	45	45	45	45	55	18	18	18	18	18	18	48	0.88
26	ASTNENode	37	48	48	49	49	49	55	81	83	83	83	83	83	83	46	0.76
27	ASTNotNode	27	27	27	27	32	32	32	33	32	32	32	32	32	32	7	0.21
28	ASTObjectArray	31	31	31	31	31	31	31	33	32	32	32	32	32	32	3	0.45
29	ASTOrNode	30	30	35	35	35	35	35	36	35	35	35	35	35	35	7	0.47
30	ASTReference	268	268	325	342	355	355	322	396	476	473	479	485	485	577	381	1.01
31	ASTSetDirective	61	61	80	80	80	80	78	102	118	118	118	118	118	118	61	0.47
32	ASTStringLiteral	66	66	65	65	65	65	64	114	150	150	150	150	150	175	113	1.27
33	ASTSubtractNode	44	44	44	45	45	45	45	55	18	18	18	18	18	18	48	0.45
34	ASTText	36	36	36	36	36	36	36	41	40	40	40	40	40	37	9	0.36
35	ASTTrue	26	26	27	27	27	27	27	28	27	27	27	27	27	27	3	0.15
36	AvalonLogSystem	73	75	67	82	78	78	76	8	8	8	8	8	8	8	99	1.18
37	BaseVisitor	189	189	189	194	194	194	192	263	263	263	263	263	263	257	84	1.05
38	ClassMap	99	99	99	242	260	260	260	192	193	193	199	199	200	200	237	3.06
39	ClasspathResourceLoader	41	41	41	41	41	41	40	50	50	50	50	50	50	50	11	0.74
40	ContentResource	31	31	31	30	30	30	51	51	55	55	55	55	55	55	26	0.57
41	DataSourceResourceLoader	164	164	164	164	166	166	166	232	243	243	243	243	243	269	105	1.18
42	Directive	36	36	36	39	39	39	40	41	41	51	51	51	51	103	67	0.5
43	Escape	46	46	46	46	46	46	46	50	50	50	50	50	50	50	4	0.32
44	FileResourceLoader	117	117	117	116	116	116	122	203	234	234	237	237	237	237	122	3.04
45	FileUtil	29	29	29	29	29	29	29	29	29	29	29	29	29	29	0	0
46	Foreach	136	151	151	155	155	155	86	230	200	200	208	208	208	264	326	4.04
47	Generator	198	198	213	213	259	259	264	275	275	275	275	275	275	275	77	0.71
48	GetExecutor	27	27	39	41	41	41	47	47	50	50	50	50	50	50	23	0.79
49	Include	88	88	100	119	119	119	114	125	131	129	129	129	129	133	59	0.54
50	InternalContextAdapterImpl	76	76	102	102	102	102	102	114	143	143	143	143	143	135	75	0.66
51	InternalContextBase	39	39	61	61	61	61	61	68	106	106	106	106	106	97	76	1.07
52	Introspector	47	47	47	55	44	44	45	56	49	49	49	49	49	49	38	1.18
53	JarHolder	99	99	94	96	96	96	92	101	106	106	106	106	106	106	25	0.8
54	JarResourceLoader	106	106	119	118	118	118	109	119	122	122	122	122	122	122	36	0.57

S.No.	Class Name	v1.0	v1.0.1	v1.1	v1.2	v1.3	v1.3.1	v1.4	v1.5	v1.6	v1.6.1	v1.6.2	v1.6.3	v1.6.4	v1.7	Change	IPC
55	JJTParserState	82	82	82	82	82	82	82	82	82	82	82	82	82	82	0	0
56	Literal	29	29	29	30	30	30	31	33	33	33	33	33	33	37	8	0.26
57	Log4JLogSystem	158	158	158	164	164	164	162	8	8	8	8	8	8	8	162	2.18
58	LogManager	31	31	31	35	79	79	79	114	150	150	150	150	150	163	132	4.95
59	Macro	102	102	102	102	97	97	116	129	97	97	97	97	97	103	75	1.1
60	MethodInvocationException	29	29	29	29	29	29	29	52	52	52	52	52	52	51	24	0.83
61	MethodMap	65	65	65	68	144	222	238	236	235	242	242	242	243	243	184	1.87
62	Node	38	38	38	40	40	40	39	41	41	41	41	41	41	42	6	0.42
63	NodeUtils	68	68	68	68	105	105	104	107	128	128	128	128	128	128	62	1.14
64	NodeViewMode	172	172	172	172	172	172	171	235	235	235	235	235	235	230	70	1.09
65	OutputWrapper	25	25	25	25	25	25	24	28	28	28	28	28	28	28	5	0.36
66	Parse	78	78	78	107	107	107	99	120	143	142	142	142	145	148	88	0.84
67	ParseException	131	131	109	110	110	110	110	111	111	111	111	111	111	111	24	1.15
68	Parser	2807	2807	2790	3014	3014	3014	2884	2974	3020	3020	3030	3030	3030	3240	727	10.44
69	ParserConstants	134	134	134	134	134	134	134	147	147	147	147	147	147	158	24	0.86
70	ParserTokenManager	3718	3718	3758	3768	3773	3773	3788	4239	4454	4454	4536	4536	4536	5622	1904	8.62
71	ParserTreeConstants	82	82	82	82	82	82	82	88	88	88	88	88	88	90	8	0.11
72	ParserVisitor	39	39	39	39	39	39	39	86	45	45	45	45	45	44	89	0.87
73	PropertyExecutor	46	46	82	103	98	98	57	66	69	69	69	69	69	69	115	4.96
74	Resource	87	87	96	92	92	92	87	87	96	96	96	96	96	96	27	0.64
75	ResourceLoader	44	44	44	47	47	47	45	74	118	118	118	118	118	118	78	0.98
76	ResourceLoaderFactory	28	28	28	28	28	28	28	28	28	28	28	28	28	26	2	0.32
77	ResourceManager	158	158	221	230	14	14	13	13	13	13	13	13	13	13	289	4.7
78	SimpleNode	196	196	196	197	197	197	195	221	235	235	235	235	235	239	47	0.81
79	SimplePool	45	45	45	45	45	45	45	49	49	49	49	49	49	49	4	0.1
80	StringUtils	267	267	285	308	310	310	310	343	343	343	343	343	343	343	76	0.54
81	Template	109	109	122	121	121	121	119	123	164	164	164	164	164	222	119	5.28
82	TexenTask	174	174	199	275	287	287	290	314	314	314	314	314	314	314	140	1.57
83	Token	22	22	19	19	19	19	19	19	19	19	19	19	19	19	3	0.33
84	TokenMgrError	87	87	72	72	72	72	72	72	72	72	72	72	72	72	15	0.74
85	TreeWalker	28	28	28	28	28	28	26	26	26	26	26	26	26	26	2	0.19
86	VelocimacroProxy	163	163	165	187	188	188	183	216	170	170	170	170	170	172	111	1.09
87	Velocity	201	201	233	227	231	231	231	257	176	176	176	176	176	170	155	2.04
88	VelocityContext	49	49	49	49	59	59	59	60	60	60	60	60	60	60	11	0.64
89	VelocityFormatter	166	166	166	185	182	182	180	180	180	180	180	180	180	180	24	1.13
90	VelocityServlet	176	176	197	195	210	210	256	249	249	249	249	249	249	249	91	0.67
91	VelocityWriter	172	172	172	172	172	172	170	161	161	161	161	161	161	161	11	0.45
92	WebMacro	146	146	176	181	181	181	181	179	179	179	179	179	179	179	37	0.58
93	XPathTool	34	34	34	27	27	27	25	25	25	25	25	25	25	25	9	0.13

A5. Experiments with DrJava

Following table tabulates LOC measure, IPC and Change metric of each class in different releases of DrJava from versions 0.9.0 through version 0.9.5.

S.No.	Class Name	v0.9.0	v0.9.2	v0.9.3	v0.9.4	v0.9.5	Change	IPC
1	AboutDialog	248	248	253	253	256	8	0.3
2	AbstractConsoleController	113	234	234	234	231	124	4.15
3	AbstractMasterJVM	126	136	137	137	179	53	0.96
4	AbstractReducedModel	131	129	129	129	129	2	0.12
5	AbstractSlaveJVM	43	47	47	47	49	6	0.25
6	ActionBracePlus	40	41	41	41	41	1	0.4
7	ActionBracePlusTest	148	147	147	147	147	1	0.14
8	ActionDoNothingTest	36	33	33	33	33	3	0.27
9	ActionStartCurrStmtPlus	25	26	26	26	26	1	0.33
10	ActionStartPrevLinePlus	31	32	32	32	32	1	0.41
11	ActionStartPrevLinePlusTest	79	79	79	79	79	0	0.37
12	ActionStartPrevStmtPlus	82	83	83	83	83	1	0.16
13	ActionStartPrevStmtPlusTest	94	91	91	91	91	3	0.34
14	ActionStartStmtOfBracePlus	31	32	32	32	32	1	0.23
15	BackSlashTest	297	297	297	297	292	5	0.5
16	BooleanOption	21	21	21	21	20	1	0.13
17	BooleanOptionComponent	24	33	33	33	33	9	0.36
18	BooleanOptionComponentTest	44	44	44	44	42	2	0.4
19	BooleanOptionTest	34	34	34	34	34	0	0
20	Brace	142	141	141	141	141	1	0.38
21	BraceTest	81	80	80	80	75	6	0.26
22	Breakpoint	74	74	74	74	70	4	0.21
23	ClasspathFilter	27	29	29	29	29	2	0.25
24	ColoringView	113	113	113	113	169	56	1.02
25	ColorOption	25	25	25	25	25	0	0
26	ColorOptionComponent	77	115	115	115	115	38	0.96
27	ColorOptionComponentTest	44	44	44	44	42	2	0.31
28	ColorOptionTest	38	38	38	38	38	0	0.55
29	CommandLineTest	231	269	269	269	269	38	0.96
30	CommentTest	120	120	120	120	115	5	0.49
31	CompilerError	102	102	102	102	102	0	0.21
32	CompilerErrorModel	212	273	274	277	280	68	0.72
33	CompilerErrorModelTest	304	288	288	288	288	16	0.7
34	CompilerErrorPanel	157	151	152	139	135	24	1.09
35	CompilerProxy	93	91	91	89	99	14	0.7
36	CompilerRegistry	112	126	126	115	108	32	0.51
37	CompilerRegistryTest	133	133	133	132	134	3	0.12
38	CompoundUndoManager	113	115	115	125	132	19	0.49

S.No.	Class Name	v0.9.0	v0.9.2	v0.9.3	v0.9.4	v0.9.5	Change	IPC
39	ConfigFileTest	33	34	34	34	34	1	0.21
40	ConfigFrame	378	534	534	540	569	191	3.76
41	ConfigPanel	88	87	87	87	85	3	0.16
42	Configuration	38	38	38	38	38	0	0.43
43	ConsoleController	140	136	136	136	134	6	0.27
44	ConsoleControllerTest	79	79	79	79	74	5	0.36
45	ConsoleDocument	174	180	180	180	185	11	1.15
46	DebugAction	61	61	61	61	53	8	0.29
47	Debugger	38	40	40	40	40	2	0.24
48	DebugPanel	604	728	728	728	742	138	4.02
49	DebugTest	1221	507	507	507	490	731	4.82
50	DebugThreadData	46	63	67	67	67	21	0.69
51	DebugWatchData	50	72	72	72	71	23	0.59
52	DefaultInteractionsModel	111	78	78	81	75	42	1.04
53	DefaultOptionMap	40	40	40	40	40	0	0
54	DefaultPlatform	25	99	99	99	102	77	0.83
55	DefinitionsDocument	1353	1396	1396	1396	698	741	3.95
56	DefinitionsDocumentTest	758	775	775	772	766	26	0.72
57	DefinitionsEditorKit	29	29	29	29	26	3	0.38
58	DefinitionsPane	657	654	662	700	771	120	3.04
59	DefinitionsPaneTest	140	199	199	298	308	168	3.75
60	DelegatingAction	110	111	111	111	111	1	0.44
61	DocumentDebugAction	77	82	82	82	80	7	0.36
62	DocumentOutputStream	31	31	31	31	30	1	0.1
63	DrJava	373	451	457	477	520	147	4.35
64	DrJavaBook	85	85	85	85	86	1	0.5
65	DrJavaBookTest	47	47	47	47	47	0	0.43
66	DummyOpenDefDoc	124	130	130	130	404	280	3.74
67	DynamicJavaAdapter	280	310	310	299	310	52	0.84
68	ErrorPanel	268	338	338	423	422	156	4.1
69	EvaluationVisitorExtension	355	380	428	434	448	93	1.06
70	EventHandlerThread	190	196	196	196	194	8	0.49
71	EventNotifier	51	34	34	34	34	17	0.95
72	EventNotifierTest	69	69	69	69	69	0	0.22
73	ExceptionResult	27	27	27	27	33	6	0.28
74	ExecJVM	101	144	144	144	145	44	0.99
75	ExecJVMTest	48	48	48	48	42	6	0.14
76	FileConfiguration	26	26	26	26	26	0	0
77	FileOption	30	30	30	30	30	0	0
78	FileOptionComponent	121	114	114	114	51	70	0.66
79	FileOptionComponentTest	45	47	47	47	46	3	0.16
80	FindReplaceDialog	366	413	413	437	456	90	0.58
81	FindReplaceMachine	233	314	314	92	89	306	1.75

S.No.	Class Name	v0.9.0	v0.9.2	v0.9.3	v0.9.4	v0.9.5	Change	IPC
82	FindReplaceMachineTest	212	313	313	347	341	141	3
83	FontChooser	173	173	173	173	170	3	0.13
84	FontOption	25	25	25	25	29	4	0.31
85	FontOptionComponent	63	70	70	70	70	7	0.29
86	FontOptionComponentTest	44	44	44	44	39	5	0.37
87	FontOptionTest	23	23	23	23	18	5	0.11
88	Free	24	24	24	24	24	0	0
89	Gap	93	93	93	93	93	0	0
90	GapTest	35	35	35	35	28	7	0.12
91	GJv6Compiler	114	112	112	112	114	4	0.4
92	GlobalIndentTest	167	179	179	179	154	37	1.2
93	GlobalModel	62	65	65	65	90	28	1.09
94	GlobalModelCompileTest	627	215	215	215	212	415	2.55
95	GlobalModelIOTest	953	982	1025	1025	1011	86	0.61
96	GlobalModelJUnitTest	307	459	460	460	493	186	5.26
97	GlobalModelListener	40	19	19	20	27	29	0.76
98	GlobalModelOtherTest	416	492	495	495	443	131	3.68
99	GlobalModelTestCase	628	672	675	690	704	76	0.96
100	HelpFrame	52	56	56	56	56	4	0.37
101	HighlightManager	146	146	146	146	146	0	0
102	HighlightStatus	27	27	27	27	27	0	0
103	History	136	135	138	186	179	59	0.94
104	HistorySaveDialog	68	45	45	45	38	30	1.05
105	HistoryTest	114	117	116	199	185	101	2.69
106	HTMLFrame	180	269	270	270	270	90	0.75
107	Indenter	69	75	75	75	75	6	0.33
108	IndentFiles	83	83	83	83	82	1	0.26
109	IndentHelperTest	305	335	335	335	335	30	1.19
110	IndentInfo	28	28	28	28	28	0	0.31
111	IndentInfoTest	54	54	54	54	62	8	0.2
112	IndentRuleQuestion	33	33	33	33	33	0	0
113	IndentRulesTestCase	35	32	32	32	43	14	0.48
114	IndentRuleWithTrace	52	54	54	54	54	2	0.48
115	IndentRuleWithTraceTest	40	37	37	37	37	3	0.23
116	IndentTest	745	713	713	713	713	32	0.68
117	InsideBlockComment	24	24	24	24	24	0	0
118	InsideDoubleQuote	32	32	32	32	32	0	1.15
119	InsideLineComment	25	25	25	25	25	0	0.46
120	InsideSingleQuote	32	32	32	32	33	1	0.23
121	IntegerOptionComponent	34	43	43	43	43	9	0.18
122	IntegerOptionComponentTest	44	44	44	44	39	5	0.35
123	IntegerOptionTest	33	33	33	33	33	0	0
124	IntegratedMasterSlaveTest	137	128	131	131	125	18	0.51

S.No.	Class Name	v0.9.0	v0.9.2	v0.9.3	v0.9.4	v0.9.5	Change	IPC
125	InteractionsController	147	318	323	344	422	275	3.02
126	InteractionsDocument	195	181	181	205	220	53	0.87
127	InteractionsDocumentTest	89	89	89	89	71	18	0.57
128	InteractionsHistoryFilter	28	28	28	28	28	0	1.15
129	InteractionsModel	262	349	349	393	403	141	3.48
130	InteractionsModelCallback	22	23	23	23	26	4	0.23
131	InteractionsModelTest	86	217	217	229	367	281	2.08
132	InteractionsPane	46	54	54	54	143	97	0.92
133	InteractionsPaneTest	149	149	149	180	216	67	0.91
134	InterpreterJVM	347	442	439	434	462	131	1.8
135	InterpreterJVMTest	113	113	113	113	113	0	0
136	Javac141FromSetLocation	32	32	32	29	28	4	0.2
137	JavacFromSetLocation	32	32	32	32	31	1	0.4
138	JavaDebugInterpreter	40	404	404	404	414	374	1.99
139	JavaDebugInterpreterTest	26	397	397	397	391	377	1.27
140	JavadocErrorPanel	116	98	99	86	69	49	1.02
141	JavadocFrame	26	97	97	97	96	72	0.48
142	JavaInterpreter	25	26	26	26	26	1	0.43
143	JavaInterpreterTest	218	275	325	344	315	155	5.26
144	JavaSourceFilter	32	32	32	32	48	16	1.08
145	JPDADebugger	1357	1433	1459	1457	1433	128	3.56
146	JSR14FromSetLocation	31	31	31	31	30	1	0.1
147	JSR14v12FromSetLocation	31	31	31	31	30	1	0.13
148	JSR14v20FromSetLocation	31	31	31	32	31	2	0.14
149	JUnitError	25	31	31	31	33	8	0.41
150	JUnitErrorModelTest	69	96	99	99	197	128	4.04
151	JUnitPanel	291	394	395	394	395	106	3.81
152	JUnitTestManager	101	148	148	148	167	66	0.49
153	JUnitTestRunner	72	69	69	69	69	3	0.46
154	KeyBindingManager	213	230	230	230	232	19	0.99
155	KeyStrokeOption	99	99	99	99	98	1	0.47
156	KeyStrokeOptionComponent	195	199	199	199	198	5	0.33
157	KeyStrokeOptionTest	98	98	98	98	97	1	0.19
158	LimitingClassLoader	31	31	31	31	31	0	0.71
159	LineEnumRule	48	75	75	75	73	29	0.95
160	MainFrameTest	332	288	288	288	326	82	1.1
161	MainJVM	359	482	487	509	516	157	1.5
162	MixedQuoteTest	65	65	65	65	60	5	0.2
163	ModelList	239	242	242	242	241	4	0.29
164	ModelListTest	237	237	237	237	229	8	0.49
165	MultiThreadedTestCase	23	25	25	25	20	7	0.1
166	NewJVMTest	255	250	250	250	251	6	0.36
167	NoCompilerAvailable	34	34	34	34	38	4	0.45

S.No.	Class Name	v0.9.0	v0.9.2	v0.9.3	v0.9.4	v0.9.5	Change	IPC
168	NoDebuggerAvailable	101	107	107	107	107	6	0.32
169	NonNegativeIntegerOption	21	21	21	21	20	1	0.25
170	OpenDefinitionsDocument	50	54	54	54	79	29	0.93
171	Option	35	37	37	37	37	2	0.1
172	OptionComponent	51	59	59	59	59	8	0.31
173	OptionConstants	247	424	452	457	506	259	4.23
174	OptionMapLoader	48	47	52	52	55	9	0.4
175	OptionMapLoaderTest	44	44	44	44	44	0	0
176	OptionParseException	21	21	21	21	21	0	0.8
177	PagePrinter	48	48	48	48	48	0	0.49
178	PendingRequestManager	87	87	87	87	83	4	0.43
179	PlatformFactory	26	26	26	26	26	0	0
180	PreventExitSecurityManager	56	56	56	56	56	0	0
181	PreviewFrame	340	340	340	340	335	5	0.42
182	QuestionBraceIsCurlyTest	92	91	91	91	91	1	0.34
183	QuestionCurrLineStartsWith	27	27	27	27	27	0	0.84
184	QuestionInsideCommentTest	51	48	48	48	48	3	0.33
185	QuestionLineContainsTest	52	49	49	49	49	3	0.42
186	QuestionNewParenPhrase	35	35	35	35	35	0	0
187	QuestionNewParenPhraseTest	80	77	77	77	77	3	0.45
188	QuestionPrevLineStartsWith	28	28	28	28	28	0	0.84
189	QuestionStartAfterOpenBrace	39	39	39	39	39	0	0
190	QuestionStartingNewStmt	31	31	31	31	31	0	0
191	QuestionStartingNewStmtTest	78	75	75	75	75	3	0.32
192	RecentFileManager	88	109	109	109	126	38	0.82
193	RecentFileManagerTest	81	119	119	127	118	55	0.86
194	ReducedModelBrace	348	347	363	363	358	22	0.99
195	ReducedModelComment	326	324	324	324	322	4	0.17
196	ReducedModelControl	224	222	222	222	228	8	0.3
197	ReducedModelDeleteTest	329	329	329	329	64	265	2.99
198	ReducedModelState	60	60	60	60	51	9	0.13
199	ReducedModelTest	799	799	799	799	778	21	0.63
200	ReducedToken	68	68	68	68	67	1	0.5
201	RMIInteractionsModel	50	61	61	64	67	17	0.68
202	SavableConfiguration	52	52	52	52	48	4	0.12
203	SimpleInteractionsModel	91	93	93	95	82	17	0.59
204	SimpleInteractionsWindow	73	82	82	86	88	15	1.18
205	SingleDisplayModel	231	11	11	11	15	224	5.55
206	SingleDisplayModelTest	253	256	262	262	239	32	0.73
207	SingleQuoteTest	297	297	297	297	275	22	0.6
208	SlaveJVMRunner	40	40	60	60	72	32	1.2
209	Step	49	54	54	54	50	9	0.43
210	StickyClassLoader	73	83	83	89	89	16	0.98

S.No.	Class Name	v0.9.0	v0.9.2	v0.9.3	v0.9.4	v0.9.5	Change	IPC
211	StickyClassLoaderTest	61	78	78	78	71	24	0.77
212	StrictURLClassLoaderTest	39	39	39	39	33	6	0.33
213	StringOptionComponent	23	34	34	34	34	11	0.75
214	StringOptionTest	24	24	24	24	24	0	0.5
215	SwingDocumentAdapter	81	84	84	84	84	3	0.15
216	SwingDocumentAdapterTest	73	73	73	73	73	0	0
217	SwingWorker	84	84	84	84	83	1	0.36
218	SyntaxErrorResult	42	42	42	42	58	16	1.18
219	TabbedPanel	48	55	55	55	53	9	0.31
220	TokenList	372	371	371	371	348	24	0.55
221	ToolbarOptionComponent	77	100	100	100	124	47	0.61
222	ToolsJarClassLoader	75	75	75	75	76	1	0.5
223	ToolsJarClassLoaderTest	21	21	21	21	18	3	0.26
224	TstampGMT	27	27	27	27	27	0	0
225	UncaughtExceptionWindow	76	77	78	78	76	4	0.18
226	UneditableTableModel	28	28	28	28	23	5	0.34
227	VectorOption	70	70	70	70	70	0	1.07
228	VectorOptionComponent	134	85	85	85	82	52	1.08
229	VectorOptionTest	72	71	71	71	70	2	0.45
231	Version	24	24	24	24	24	0	0.54

A6. Experiments with jEdit

Following table tabulates LOC measure, IPC and Change metric of each class in different releases of jEdit from versions 4.3.0 through version 5.0.0.

S. No.	Class Name	v4.3.0	v4.3.1	v4.3.2	v4.3.3	v4.4.1	v4.4.2	v4.5.0	v4.5.1	v4.5.2	v5.0.0	Change	IPC
1	AbbrevEditor	170	170	170	170	170	170	170	170	170	170	0	0
2	Abbrevs	400	400	400	400	400	400	400	400	400	400	0	1.16
3	AbbrevsOptionPane	385	385	385	385	385	385	385	385	385	385	0	0
4	AboutDialog	254	254	254	253	261	261	261	261	261	260	10	1.56
5	AbstractContextOptionPane	219	219	219	219	219	219	229	229	229	229	10	2.64
6	AbstractInputHandler	220	220	220	220	220	220	220	220	220	223	3	1.94
7	AbstractOptionPane	196	196	196	196	196	196	196	196	196	183	13	2.23
8	ActionBar	445	445	445	445	435	435	435	435	435	435	10	2.8
9	ActionListHandler	128	128	128	128	128	128	128	128	128	128	0	0
10	ActionSet	109	109	109	109	109	109	109	109	109	111	2	0.98
11	AddAbbrevDialog	92	92	92	92	92	92	92	92	92	92	0	0.94
12	AllBufferSet	50	50	50	50	50	50	50	50	50	50	0	0
13	Anchor	63	63	63	63	63	63	63	63	63	95	32	2.81
14	AnimatedIcon	72	72	72	72	72	72	72	72	72	72	0	1.16
15	AntiAlias	47	47	47	47	47	47	67	67	67	67	20	2.02

S. No.	Class Name	v4.3.0	v4.3.1	v4.3.2	v4.3.3	v4.4.1	v4.4.2	v4.5.0	v4.5.1	v4.5.2	v5.0.0	Change	IPC
16	AppearanceOptionPane	192	192	192	192	199	199	205	205	205	190	28	0.6
17	AutoDetection	123	123	123	123	123	123	123	123	123	123	0	0
18	Autosave	55	55	55	55	55	55	55	55	55	55	0	0
19	BeanShell	281	281	281	281	281	281	257	257	257	257	24	0.38
20	BeanShellAction	122	122	122	122	122	122	122	122	122	122	0	0
21	BeanShellFacade	183	183	183	183	188	188	188	188	188	188	5	1.3
22	BlockNameSpace	54	54	54	54	54	54	54	54	54	54	0	1.15
23	BoyerMooreSearchMatcher	146	146	146	146	147	147	164	164	164	167	21	1.45
24	BracketIndentRule	148	148	148	148	148	148	119	119	119	119	29	0.33
25	BrowserColorsOptionPane	304	304	304	304	304	304	304	304	304	304	0	0
26	BrowserCommandsMenu	227	227	227	227	227	227	251	251	251	260	33	1.4
27	BrowserOptionPane	111	111	111	111	111	111	111	111	111	111	0	0
28	BrowserView	552	552	552	552	564	564	568	568	568	568	16	2.82
29	BSHAllocationExpression	193	193	193	193	193	193	193	193	193	193	0	0
30	BSHAmbiguousName	57	57	57	57	57	57	57	57	57	57	0	0.65
31	BSHArrayDimensions	68	68	68	68	68	68	68	68	68	68	0	1.11
32	BSHArrayInitializer	76	76	76	76	76	76	76	76	76	76	0	0.41
33	BSHAssignment	108	108	108	108	108	108	108	108	108	108	0	0
34	BSHBinaryExpression	89	89	89	89	89	89	89	89	89	89	0	0
35	BSHBlock	79	79	79	79	79	79	79	79	79	79	0	0.6
36	BSHClassDeclaration	46	46	46	46	46	46	46	46	46	46	0	1.03
37	BshClassLoader	66	66	66	66	66	66	66	66	66	66	0	0.43
38	BshClassManager	297	297	297	297	297	297	297	297	297	297	0	0
39	BshClassPath	534	534	534	534	534	534	534	534	534	534	0	1.16
40	BSHEnhancedForStatement	78	78	78	78	78	78	78	78	78	78	0	0.95
41	BSHFormalParameter	26	26	26	26	26	26	26	26	26	26	0	0
42	BSHFormalParameters	57	57	57	57	57	57	57	57	57	57	0	0
43	BSHForStatement	68	68	68	68	68	68	68	68	68	68	0	0
44	BSHIfStatement	38	38	38	38	38	38	38	38	38	38	0	0
45	BSHImportDeclaration	42	42	42	42	42	42	42	42	42	42	0	0
46	BSHLiteral	81	81	81	81	81	81	81	81	81	81	0	0
47	BshMethod	222	222	222	222	222	222	222	222	222	222	0	0.65
48	BSHMethodDeclaration	95	95	95	95	95	95	95	95	95	95	0	0.72
49	BSHMethodInvocation	44	44	44	44	44	44	44	44	44	44	0	0
50	BSHPrimaryExpression	54	54	54	54	54	54	54	54	54	54	0	0.64
51	BSHPrimarySuffix	185	185	185	185	185	185	185	185	185	185	0	0
52	BSHReturnType	26	26	26	26	26	26	26	26	26	26	0	0
53	BSHSwitchStatement	76	76	76	76	76	76	76	76	76	76	0	0
54	BSHTryStatement	96	96	96	96	96	96	96	96	96	96	0	0
55	BSHType	112	112	112	112	112	112	112	112	112	112	0	0
56	BSHTypedVariableDeclaration	56	56	56	56	56	56	56	56	56	56	0	0
57	BSHUnaryExpression	80	80	80	80	80	80	80	80	80	80	0	0
58	BSHVariableDeclarator	28	28	28	28	28	28	28	28	28	28	0	0

S. No.	Class Name	v4.3.0	v4.3.1	v4.3.2	v4.3.3	v4.4.1	v4.4.2	v4.5.0	v4.5.1	v4.5.2	v5.0.0	Change	IPC
59	BSHWhileStatement	48	48	48	48	48	48	48	48	48	48	0	0
60	Buffer	1318	1318	1390	1390	1346	1346	1308	1308	1308	1385	231	5.34
61	BufferAutosaveRequest	60	60	60	60	60	60	64	64	64	64	4	0.35
62	BufferChanging	23	23	23	23	23	23	23	23	23	23	0	1.12
63	BufferHandler	303	303	303	303	303	304	304	304	304	289	16	1.74
64	BufferHistory	336	336	336	336	336	353	356	356	356	356	20	0.57
65	BufferInsertRequest	74	74	74	74	74	74	74	74	74	74	0	1.1
66	BufferIORequest	292	292	292	292	292	292	284	284	284	284	8	0.88
67	BufferListSet	83	83	83	83	83	83	83	83	83	126	43	1.72
68	BufferLoadRequest	288	288	288	288	288	288	288	288	288	288	0	0.97
69	BufferOptionPane	246	246	246	246	246	246	255	255	255	261	15	0.5
70	BufferOptions	67	67	67	67	67	67	67	67	67	67	0	1.12
71	BufferPrintable	304	304	304	304	304	304	305	305	305	305	1	1.96
72	BufferPrinter1_3	67	67	67	67	67	67	67	67	67	67	0	0.34
73	BufferPrinter1_4	131	131	131	131	131	131	131	131	131	131	0	0
74	BufferSaveRequest	143	143	143	143	144	144	151	151	151	159	16	1.88
75	BufferSegment	74	74	74	74	74	74	74	74	74	74	0	0
76	BufferSet	258	258	258	258	241	241	241	241	241	241	17	0.78
77	BufferSetManager	225	225	225	225	262	280	280	280	280	280	55	2.8
78	BufferSetWidgetFactory	113	113	113	113	104	104	104	104	104	104	9	1.07
79	BufferSwitcher	76	76	76	76	86	86	90	90	90	86	18	1.89
80	BufferUpdate	42	42	42	42	42	42	42	42	42	42	0	0
81	ByteVector	144	144	144	144	144	144	144	144	144	144	0	0.51
82	CallStack	62	62	62	62	62	62	62	62	62	62	0	0.78
83	Capabilities	52	52	52	52	52	52	52	52	52	52	0	0
84	CharsetEncoding	34	34	34	34	34	34	34	34	34	34	0	0.53
85	Chunk	220	220	224	224	351	351	351	341	341	551	351	5.56
86	ChunkCache	494	494	494	494	494	494	496	496	496	496	2	1.29
87	ClassGenerator	33	33	33	33	33	33	33	33	33	33	0	0
88	ClassGeneratorImpl	228	228	228	228	228	228	228	228	228	228	0	0.75
89	ClassGeneratorUtil	707	707	707	707	707	707	707	707	707	707	0	0
90	ClassManagerImpl	270	270	270	270	270	270	270	270	270	270	0	1.06
91	ClassVisitor	21	21	21	21	21	21	21	21	21	21	0	0
92	ClassWriter	398	398	398	398	398	398	398	398	398	398	0	0
93	ClockWidgetFactory	92	92	92	92	92	92	92	92	92	92	0	0.85
94	CloseBracketIndentRule	112	112	112	112	112	112	107	107	107	107	5	1.52
95	CloseDialog	193	193	193	193	193	193	193	193	193	193	0	0.28
96	CodeVisitor	25	25	25	25	25	25	25	25	25	25	0	0
97	CodeWriter	968	968	968	968	968	968	968	968	968	968	0	0
98	CollectionIterator	28	28	28	28	28	28	28	28	28	28	0	0.55
99	CollectionManager	93	93	93	93	93	93	93	93	93	93	0	0.55
100	CollectionManagerImpl	31	31	31	31	31	31	31	31	31	31	0	0
101	ColorWellButton	134	134	134	134	134	134	77	77	77	77	57	1.83

S. No.	Class Name	v4.3.0	v4.3.1	v4.3.2	v4.3.3	v4.4.1	v4.4.2	v4.5.0	v4.5.1	v4.5.2	v5.0.0	Change	IPC
102	CommandLineReader	44	44	44	44	44	44	44	44	44	44	0	0
103	CompleteWord	377	377	377	377	377	377	376	376	376	376	1	1.75
104	CompletionPopup	313	313	313	313	334	334	334	334	334	344	31	1.89
105	Constants	182	182	182	182	182	182	182	182	182	182	0	0
106	ContentManager	147	147	147	147	147	147	147	147	147	147	0	0
107	ContextAddDialog	155	155	155	155	171	171	173	173	173	173	18	0.7
108	ContextOptionPane	25	25	25	25	18	18	20	20	20	34	23	1.08
109	CopyFileWorker	30	30	30	30	42	42	42	42	42	182	152	1.87
110	CurrentBufferSet	28	28	28	28	28	28	28	28	28	39	11	1.14
111	DeepIndentRule	160	160	160	160	160	160	160	160	160	160	0	1.15
112	DefaultInputHandler	143	143	143	143	143	143	143	143	143	143	0	1.19
113	DefaultTokenHandler	57	57	57	57	57	57	57	57	57	57	0	0
114	DelayedEvalBshMethod	48	48	48	48	48	48	48	48	48	48	0	0.5
115	dir	72	72	72	72	72	72	72	72	72	72	0	0
116	DirectoryListSet	108	108	108	108	108	108	108	108	108	108	0	0.27
117	DirectoryProvider	110	110	110	110	110	110	110	110	110	110	0	0
118	DiscreteFilesClassLoader	36	36	36	36	36	36	36	36	36	36	0	0
119	DisplayManager	581	581	581	581	581	611	632	644	644	626	81	1.86
120	DisplayTokenHandler	146	146	146	146	145	145	148	148	148	478	334	4.42
121	DockableLayout	355	355	355	355	355	355	350	350	350	350	5	1.01
122	DockablePanel	233	233	233	233	233	233	233	233	233	233	0	0.37
123	DockableWindowFactory	401	401	401	401	401	401	401	401	401	407	6	0.43
124	DockableWindowManager	437	437	437	437	435	435	435	435	435	439	6	0.8
125	DockableWindowManagerImpl	768	768	768	768	768	768	768	768	768	773	5	0.86
126	DockableWindowUpdate	34	34	34	34	34	34	34	34	34	34	0	0
127	DockingLayoutManager	241	241	241	241	241	241	241	241	241	241	0	0
128	DockingOptionPane	304	304	304	304	304	304	304	304	304	333	29	0.38
129	EditAbbrevDialog	115	115	115	115	115	115	115	115	115	115	0	0
130	EditingOptionPane	336	336	336	336	336	336	348	348	348	413	77	1.63
131	EditorExitRequested	23	23	23	23	23	23	23	23	23	23	0	0.78
132	EditPaneUpdate	31	31	31	31	31	31	31	31	31	31	0	1.17
133	EncodingServer	83	83	83	83	83	83	83	83	83	83	0	0.51
134	EncodingsOptionPane	164	164	164	164	132	132	124	124	124	124	40	1.45
135	EncodingWidgetFactory	50	50	50	50	50	50	50	50	50	50	0	0
136	EncodingWithBOM	121	121	121	121	121	121	121	121	121	121	0	0
137	EnhancedButton	69	69	69	69	69	69	69	69	69	69	0	0
138	EnhancedCheckBoxMenuItem	120	120	120	120	131	131	131	131	131	145	25	1.32
139	EnhancedDialog	121	121	121	121	121	121	121	121	121	133	12	2.23
140	EnhancedMenu	120	120	120	120	119	119	119	119	119	119	1	0.9
141	EnhancedMenuItem	122	122	128	128	132	132	132	132	132	146	24	2.45
142	ErrorListDialog	151	151	151	151	151	151	151	151	157	162	11	1.28
143	ErrorsWidgetFactory	221	221	221	221	221	234	234	234	234	239	18	1.9
144	EvalError	88	88	88	88	88	88	88	88	88	88	0	0.41

S. No.	Class Name	v4.3.0	v4.3.1	v4.3.2	v4.3.3	v4.4.1	v4.4.2	v4.5.0	v4.5.1	v4.5.2	v5.0.0	Change	IPC
145	ExplicitFoldHandler	51	51	51	51	51	51	51	51	51	51	0	0
146	ExtendedGridLayout	796	796	796	796	796	796	796	796	796	796	0	1.04
147	ExtensionManager	120	120	120	120	120	120	120	120	120	120	0	0
148	ExternalNameSpace	106	106	106	106	106	106	106	106	106	106	0	0.35
149	FavoritesProvider	68	68	68	68	68	68	69	69	69	69	1	1.71
150	FavoritesVFS	132	132	132	132	132	132	171	171	171	174	42	0.91
151	FileCellRenderer	199	199	199	199	199	199	208	208	208	208	9	0.7
152	FilePropertiesDialog	241	241	241	241	248	248	268	268	268	268	27	1.51
153	FileRootsVFS	106	106	106	106	106	106	106	106	106	107	1	0.4
154	FilesChangedDialog	327	327	327	327	327	327	331	331	331	331	4	1.35
155	FileVFS	443	443	443	443	448	448	447	447	447	424	29	1
156	FilteredListModel	157	157	157	157	157	157	157	157	157	157	0	0
157	FilteredTableModel	150	150	150	150	150	150	150	150	150	154	4	1.22
158	FirewallOptionPane	78	78	78	78	78	78	78	78	78	78	0	0.89
159	FirstLine	263	263	263	263	263	263	263	263	263	395	132	2.31
160	FloatingWindowContainer	149	149	149	149	149	149	149	149	149	149	0	0
161	FoldHandler	47	47	47	47	47	47	45	45	45	45	2	1.9
162	FoldWidgetFactory	50	50	50	50	50	50	50	50	50	50	0	0
163	FontSelector	353	353	353	353	99	99	99	99	99	99	254	1.17
164	GeneralOptionPane	180	180	180	180	180	180	157	157	157	198	64	1.75
165	GlobalOptions	76	76	76	76	76	76	66	66	66	66	10	1.6
166	GlobVFSFileFilter	51	51	51	51	51	51	51	51	51	51	0	1.04
167	GrabKeyDialog	370	370	370	370	366	366	372	372	372	373	11	1.24
168	Gutter	759	759	759	759	763	763	767	767	767	769	10	2.34
169	GutterOptionPane	281	281	281	281	281	281	262	262	262	262	19	1.78
170	HelpHistoryModel	171	171	171	171	171	171	171	171	171	171	0	0.93
171	HelpIndex	280	280	280	280	280	280	280	280	280	280	0	0
172	HelpSearchPanel	223	223	223	223	223	223	223	223	223	223	0	0
173	HelpTOCPanel	330	330	345	345	347	347	351	351	351	351	21	0.62
174	HelpViewer	401	401	401	401	411	411	411	411	411	420	19	0.69
175	HistoryButton	124	124	124	124	124	124	124	124	124	124	0	0.66
176	HistoryModel	84	84	84	84	104	104	139	139	139	139	55	2.32
177	HistoryText	248	248	248	248	248	248	248	248	248	248	0	0.79
178	HistoryTextArea	119	119	119	119	119	119	119	119	119	119	0	0
179	HistoryTextField	225	225	225	225	226	226	226	226	226	226	1	1.38
180	HyperSearchFileNode	55	55	55	55	55	55	55	55	55	55	0	0
181	HyperSearchFolderNode	29	29	29	29	29	29	29	29	29	29	0	0.54
182	HyperSearchOperationNode	170	170	170	170	170	170	173	173	173	173	3	1.23
183	HyperSearchRequest	224	224	224	224	221	221	223	223	223	223	5	0.62
184	HyperSearchResult	138	138	138	138	181	181	181	181	181	181	43	1.61
185	HyperSearchResults	901	901	901	901	861	861	864	864	864	881	60	1.19
186	IndentAction	126	126	126	126	126	126	126	126	126	126	0	0.7
187	IndentFoldHandler	68	68	68	68	68	68	68	68	68	68	0	1.2

S. No.	Class Name	v4.3.0	v4.3.1	v4.3.2	v4.3.3	v4.4.1	v4.4.2	v4.5.0	v4.5.1	v4.5.2	v5.0.0	Change	IPC
188	IndentRuleFactory	59	59	59	59	59	59	59	59	59	59	0	0
189	InputHandler	334	334	334	334	334	334	329	329	329	329	5	0.33
190	InputMethodSupport	208	208	208	208	208	208	208	208	208	208	0	0
191	InstallPanel	936	936	936	936	935	935	1087	1087	1087	1085	155	5.77
192	IntegerArray	44	44	44	44	44	44	44	44	44	44	0	0.95
193	Interpreter	628	628	628	628	628	628	628	628	628	628	0	0.38
194	IOProgressMonitor	145	145	145	145	147	147	147	147	147	147	2	1.93
195	IOUtilities	167	167	167	167	168	168	168	168	168	168	1	0.31
196	Item	98	98	98	98	98	98	98	98	98	98	0	0.59
197	JARClassLoader	380	380	380	380	381	381	368	368	368	368	14	1.78
198	JavaCharStream	447	447	447	447	447	447	447	447	447	447	0	0
199	JCheckBoxList	245	245	245	245	245	245	245	245	245	245	0	0.76
200	jEdit	2774	2774	2774	2774	2793	2817	2820	2820	2820	2976	202	6.04
201	JEditAbstractEditAction	34	34	34	34	34	34	34	34	34	34	0	0
202	JEditActionContext	78	78	78	78	78	78	78	78	78	78	0	1.09
203	JEditActionSet	184	184	184	184	179	179	179	179	179	179	5	1.15
204	JEditBeanShellAction	122	122	122	122	122	122	122	122	122	122	0	1.14
205	JEditBuffer	1737	1737	1678	1678	1699	1699	1794	1794	1794	1807	188	2.8
206	JEditHistoryModelSaver	156	156	156	156	156	156	156	156	156	156	0	0.6
207	JEditKillRing	142	142	142	142	142	153	164	164	164	164	22	2.33
208	JEditMode	55	55	55	55	82	82	82	82	82	82	27	0.43
209	JEditRegisterSaver	133	133	133	133	133	133	133	133	133	133	0	0.67
210	JEditTextArea	339	375	375	375	375	375	300	306	306	294	129	4.42
211	JJTParserState	81	81	81	81	81	81	81	81	81	81	0	1.12
212	JThis	159	159	159	159	159	159	159	159	159	159	0	0
213	KeyEventTranslator	370	370	370	370	365	365	355	355	355	402	62	2
214	KeyEventWorkaround	285	285	285	285	285	285	285	285	285	285	0	0.65
215	KeywordMap	128	128	128	128	128	128	128	128	128	128	0	0
216	KillRing	193	193	193	193	193	193	199	199	199	199	6	0.82
217	Label	103	103	103	103	103	103	103	103	103	103	0	0
218	LHS	158	158	158	158	158	158	158	158	158	158	0	0
219	LineManager	240	240	240	240	240	240	240	240	240	253	13	0.86
220	LineSepWidgetFactory	68	68	68	68	68	68	68	68	68	68	0	1.03
221	ListModelEditor	100	100	100	100	100	100	100	100	100	100	0	0.69
222	Log	386	386	386	386	386	386	387	387	387	387	1	0.36
223	LogViewer	298	298	298	298	438	438	456	456	456	456	158	1.48
224	Macros	604	604	604	606	615	617	617	617	617	716	112	2.24
225	MacrosProvider	53	53	53	53	53	53	63	63	63	63	10	2.79
226	ManagePanel	1073	1073	1073	1073	1063	1064	1080	1080	1080	1084	31	1.95
227	Marker	43	43	43	43	43	43	43	43	43	43	0	0
228	MarkersProvider	125	125	125	125	125	125	125	125	125	125	0	0
229	MarkersSaveRequest	86	86	86	86	86	86	86	86	86	86	0	0
230	MarkerViewer	207	207	207	207	207	207	207	207	207	221	14	0.42

S. No.	Class Name	v4.3.0	v4.3.1	v4.3.2	v4.3.3	v4.4.1	v4.4.2	v4.5.0	v4.5.1	v4.5.2	v5.0.0	Change	IPC
231	MemoryStatusWidgetFactory	166	166	166	166	166	166	166	166	166	168	2	0.75
232	MenuItemTextComparator	26	26	26	26	26	26	26	26	26	26	0	0
233	MirrorList	117	117	117	117	127	127	127	127	127	127	10	1.17
234	MirrorListHandler	98	98	98	98	98	98	98	98	98	98	0	0.27
235	MiscUtilities	1013	1013	1013	1013	1002	1006	798	798	798	930	355	1.17
236	Mode	261	261	261	261	272	272	293	293	293	293	32	1.28
237	ModeCatalogHandler	63	63	63	63	61	61	61	61	61	61	2	1.68
238	ModeProvider	129	129	129	129	183	183	189	189	189	161	88	1.73
239	ModeWidgetFactory	50	50	50	50	50	50	50	50	50	50	0	0.22
240	Modifiers	67	67	67	67	67	67	67	67	67	67	0	0
241	MouseActions	29	29	29	29	29	29	29	29	29	29	0	0
242	MouseHandler	115	115	115	115	115	115	118	118	118	118	3	0.38
243	MouseOptionPane	98	98	98	98	98	98	98	98	98	93	5	0.37
244	MultiSelectWidgetFactory	65	65	65	65	71	71	71	71	71	71	6	0.24
245	Name	541	541	541	541	541	541	541	541	541	541	0	0
246	NameSpace	811	811	811	811	811	811	811	811	811	811	0	0
247	NumericTextField	28	28	28	28	28	28	28	28	28	28	0	0
248	OpenBracketIndentRule	54	54	54	54	54	54	54	54	54	54	0	0.54
249	OperatingSystem	226	226	226	226	226	226	212	212	212	209	17	2.66
250	OptionGroup	108	108	108	108	108	108	108	108	108	108	0	0.66
251	OptionsDialog	532	532	532	532	532	532	536	536	536	544	12	1.62
252	OverwriteWidgetFactory	63	63	63	63	73	73	73	73	73	73	10	2.85
253	PanelWindowContainer	713	713	713	713	713	713	713	713	713	713	0	0
254	ParseException	149	149	149	149	149	149	149	149	149	149	0	1.05
255	Parser	5319	5319	5319	5319	5319	5319	5319	5319	5319	5319	0	0
256	ParserConstants	269	269	269	269	269	269	269	269	269	269	0	0.2
257	ParserRule	280	280	280	280	283	283	289	289	289	289	9	0.87
258	ParserRuleSet	256	256	256	256	256	256	246	246	246	246	10	2.99
259	ParserTokenManager	2099	2099	2099	2099	2099	2099	2099	2099	2099	2099	0	0.31
260	ParserTreeConstants	82	82	82	82	82	82	82	82	82	82	0	0.36
261	PasteFromListDialog	179	179	179	179	179	179	222	222	222	222	43	2.79
262	PatternSearchMatcher	114	120	120	120	120	120	131	131	131	131	17	0.36
263	PerspectiveManager	248	248	248	248	268	268	270	270	270	270	22	1.12
264	PluginDetailPanel	53	53	53	58	58	58	58	58	58	58	5	0.96
265	PluginJAR	1246	1246	1246	1246	1252	1252	1260	1260	1266	1401	155	3.88
266	PluginList	419	419	419	419	416	416	443	443	443	492	79	2.16
267	PluginListHandler	207	207	207	207	207	207	207	207	207	207	0	0.56
268	PluginManager	265	265	265	265	218	223	223	223	223	223	52	0.75
269	PluginManagerOptionPane	269	269	269	269	287	287	287	287	287	287	18	1.55
270	PluginManagerProgress	145	145	145	145	149	149	138	137	137	137	16	1.22
271	PluginOptions	93	93	93	93	93	93	80	80	80	80	13	2.81
272	PluginResURLConnection	93	93	93	93	93	93	93	93	93	100	7	1.48
273	PluginsProvider	138	138	138	138	138	138	112	112	112	112	26	2.63

S. No.	Class Name	v4.3.0	v4.3.1	v4.3.2	v4.3.3	v4.4.1	v4.4.2	v4.5.0	v4.5.1	v4.5.2	v5.0.0	Change	IPC
274	PluginUpdate	36	36	36	36	47	47	47	47	47	47	11	2.04
275	PositionManager	125	125	125	125	125	125	125	125	125	125	0	0.4
276	Primitive	716	716	716	716	716	716	716	716	716	716	0	0
277	PrintOptionPane	71	71	71	71	71	71	71	71	71	71	0	0.81
278	PropertiesBean	181	181	181	181	181	181	181	181	181	181	0	0
279	PropertiesChanging	21	21	21	21	21	21	21	21	21	21	0	0.99
280	PropertyManager	123	123	123	123	123	123	123	123	123	152	29	2.43
281	RangeMap	350	350	350	350	350	350	350	350	350	350	0	0
282	RecentDirectoriesProvider	92	92	92	92	92	92	92	92	92	92	0	0.75
283	RecentFilesProvider	126	126	126	126	142	142	143	143	143	143	17	1.91
284	RectSelectWidgetFactory	65	65	65	65	73	73	73	73	73	73	8	1.57
285	Reflect	598	598	598	598	598	598	598	598	598	598	0	0
286	ReflectManager	27	27	27	27	27	27	27	27	27	27	0	0
287	RegexEncodingDetector	50	50	50	55	55	55	55	55	55	55	5	1.34
288	RegexpIndentRule	109	109	109	109	109	109	103	103	103	103	6	0.4
289	RegisterChanged	25	25	25	25	25	25	25	25	25	25	0	0
290	Registers	370	370	370	370	576	576	576	576	576	538	244	6.34
291	RegisterViewer	310	310	310	310	310	310	310	310	310	343	33	1.26
292	ReloadWithEncodingProvider	101	102	102	102	102	102	102	102	102	106	5	0.4
293	Remote	126	126	126	126	126	126	126	126	126	126	0	0
294	ReverseCharSequence	37	37	37	37	37	37	37	37	37	37	0	1.16
295	RolloverButton	78	78	78	78	78	78	78	78	78	78	0	0
296	Roster	346	346	346	346	346	346	346	346	343	348	8	0.34
297	SaveBackupOptionPane	124	124	124	124	126	126	128	128	128	135	11	1.14
298	ScreenLineManager	76	76	76	76	76	76	76	76	76	76	0	0
299	ScrollLayout	149	149	149	149	149	149	149	149	149	149	0	0.87
300	ScrollLineCount	30	30	30	30	30	30	30	30	30	90	60	1.35
301	SearchAndReplace	874	874	874	874	886	886	910	910	910	915	41	2.84
302	SearchBar	338	338	338	338	338	338	350	350	350	360	22	1.09
303	SearchDialog	794	794	794	794	803	803	812	812	812	816	22	2.15
304	SegmentBuffer	32	32	32	32	32	32	32	32	32	32	0	0.46
305	SegmentCharSequence	46	46	46	46	46	46	35	35	35	35	11	0.62
306	Selection	499	499	499	499	499	499	499	499	499	507	8	0.72
307	SelectionManager	275	275	275	275	275	275	275	275	275	275	0	0
308	SelectLineRange	124	124	124	124	124	124	124	124	124	124	0	0
309	ServiceListHandler	94	94	94	94	94	94	94	94	94	94	0	0.46
310	ServiceManager	162	162	162	162	170	170	170	170	170	174	12	2.95
311	SettingsReloader	56	56	56	56	54	54	54	54	54	54	2	1.51
312	SettingsXML	84	84	84	84	84	84	84	84	84	84	0	0.77
313	ShapedFoldPainter	46	46	46	46	47	47	47	47	47	47	1	0.56
314	ShortcutPrefixActiveEvent	45	45	45	45	45	45	45	45	45	45	0	0
315	ShortcutsOptionPane	365	365	365	365	365	377	382	382	382	601	236	4.7
316	SimpleNode	106	106	106	106	106	106	106	106	106	106	0	0

S. No.	Class Name	v4.3.0	v4.3.1	v4.3.2	v4.3.3	v4.4.1	v4.4.2	v4.5.0	v4.5.1	v4.5.2	v5.0.0	Change	IPC
317	SplashScreen	142	142	142	142	142	142	142	142	142	144	2	1.24
318	StandaloneTextArea	437	437	437	437	447	447	447	447	447	449	12	1.67
319	StandardUtilities	442	442	442	442	473	473	473	473	473	473	31	0.69
320	StatusBar	343	343	347	347	349	349	349	349	349	354	11	1.94
321	StatusBarOptionPane	367	367	367	367	379	379	429	451	451	447	88	1.92
322	StringList	77	77	77	77	73	73	73	73	73	73	4	0.22
323	StringUtil	58	58	58	58	58	58	58	58	58	58	0	0
324	StructureMatcher	152	152	152	152	152	152	152	152	152	152	0	0.95
325	StyleEditor	126	126	126	126	169	169	169	169	169	169	43	0.64
326	SyntaxHiliteOptionPane	213	213	213	213	213	213	213	213	213	213	0	0
327	SyntaxStyle	27	27	27	27	27	27	27	27	27	27	0	0
328	SyntaxUtilities	26	26	26	26	26	26	26	26	26	26	0	0.82
329	SyntaxUtilities	137	137	137	137	137	137	138	138	138	138	1	0.39
330	TargetError	74	74	74	74	74	74	74	74	74	74	0	1.02
331	TextArea	4048	4048	4051	4052	4063	4073	4143	4338	4338	4340	292	1.4
332	TextAreaBorder	25	25	25	25	25	25	25	25	25	25	0	0.77
333	TextAreaDialog	81	81	81	81	81	81	81	81	81	81	0	0
334	TextAreaDropHandler	57	57	57	57	54	54	54	54	54	54	3	0.91
335	TextAreaExtension	30	30	30	30	30	30	30	30	30	30	0	0.94
336	TextAreaInputHandler	245	245	245	245	245	245	245	245	245	241	4	1.71
337	TextAreaMouseHandler	450	450	450	450	447	447	474	474	474	474	30	2.13
338	TextAreaOptionPane	177	177	177	177	336	336	350	350	350	376	199	2.73
339	TextAreaPainter	698	698	698	698	865	865	847	847	847	847	185	5.62
340	TextAreaTransferHandler	364	364	364	375	370	370	379	379	379	379	25	1.55
341	TextUtilities	606	606	606	606	627	627	632	632	632	632	26	1.59
342	This	130	130	130	130	130	130	130	130	130	130	0	0.85
343	TipOfTheDay	105	105	105	105	105	105	105	105	105	105	0	0.28
344	Token	79	79	79	79	79	79	79	79	79	79	0	0
345	TokenMarker	674	674	674	674	691	691	734	740	740	740	66	0.33
346	TokenMgrError	72	72	72	72	72	72	72	72	72	72	0	0
347	ToolBarManager	70	70	70	70	70	70	70	70	70	70	0	0
348	ToolBarOptionPane	605	605	605	605	613	613	613	613	613	613	8	1.76
349	TriangleFoldPainter	40	40	40	40	41	41	41	41	41	41	1	0.81
350	Type	303	303	303	303	303	303	303	303	303	303	0	0.9
351	Types	253	253	253	253	253	253	253	253	253	253	0	0.23
352	UndoManager	347	347	347	347	346	346	346	346	346	346	1	0.63
353	UrlVFS	51	51	51	51	51	51	64	64	64	64	13	2.85
354	UtilEvalError	24	24	24	24	24	24	24	24	24	24	0	0
355	UtilTargetError	21	21	21	21	21	21	21	21	21	21	0	0.85
356	Variable	74	74	74	74	74	74	74	74	74	74	0	0
357	VFS	522	522	522	522	540	540	498	498	498	600	162	3.76
358	VFSBrowser	1524	1524	1524	1524	1509	1513	1529	1529	1528	1557	65	0.63
359	VFSDirectoryEntryTable	527	527	527	527	530	530	530	530	531	531	4	1.94

S. No.	Class Name	v4.3.0	v4.3.1	v4.3.2	v4.3.3	v4.4.1	v4.4.2	v4.5.0	v4.5.1	v4.5.2	v5.0.0	Change	IPC
360	VFSDirectoryEntryTableModel	288	288	288	288	288	288	288	288	288	288	0	0.3
361	VFSFile	291	291	291	291	301	301	292	292	292	292	19	1.39
362	VFSFileChooserDialog	448	448	448	448	451	451	451	451	451	469	21	0.85
363	VFSFileNameField	214	214	214	214	214	214	214	214	214	214	0	0
364	VFSManager	248	248	248	248	254	254	199	199	267	275	137	6.6
365	VFSPathSelected	26	26	26	26	26	26	26	26	26	26	0	1.16
366	VFSUpdate	21	21	21	21	21	21	21	21	21	21	0	0
367	View	1447	1446	1446	1455	1467	1467	1460	1460	1460	1474	43	0.88
368	ViewOptionPane	197	197	197	197	197	197	204	204	204	204	7	0.42
369	ViewUpdate	29	29	29	29	29	29	29	29	29	29	0	0
370	WhitespaceRule	32	32	32	32	32	32	32	32	32	32	0	0
371	WorkRequest	61	61	61	61	64	64	64	64	64	64	3	1.31
372	WorkThread	149	149	149	149	150	150	150	150	150	150	1	1.35
373	WorkThreadPool	293	293	293	293	291	291	291	291	291	291	2	0.95
374	WrapWidgetFactory	66	66	66	66	74	74	74	74	74	84	18	2.54
375	XMLEncodingDetector	56	56	56	56	56	56	56	56	56	56	0	0
376	XMLUtilities	112	112	112	112	112	112	104	104	104	104	8	0.21
377	XModeHandler	597	597	597	597	597	597	603	603	603	608	11	1.08
378	XThis	87	87	87	87	87	87	87	87	87	87	0	0

A7. Experiments with jFlex

Following table tabulates LOC measure, IPC and Change metric of each class in different releases of jFlex from versions 1.3.0 through version 1.4pre5.

S.No.	Class Name	v1.3.0	v1.3.1	v1.3.2	v1.3.3	v1.3.4	v1.3.5	v1.4pre1	v1.4pre3	v1.4pre4	v1.4pre5	Change	IPC
1	Action	30	30	30	31	31	31	31	31	38	38	8	1.76
2	GeneratorThread	31	31	31	31	31	31	31	31	40	40	9	0.24
3	GridPanel	106	106	106	106	106	106	106	106	106	106	0	0
4	IntCharSet	216	216	216	216	216	216	216	216	260	260	44	1.09
5	IntPair	22	22	22	22	22	22	22	22	22	22	0	0.31
6	LexicalStates	22	22	22	22	30	30	30	30	30	30	8	1.37
7	LexParse	1383	1382	1382	1382	1382	1417	1417	1391	1379	1379	74	2.39
8	LexScan	2160	2169	2239	2498	2540	2545	2545	2532	2289	2289	641	2.9
9	Macros	70	70	70	70	70	70	70	80	80	80	10	0.35
10	Main	195	213	213	237	240	240	240	240	226	226	59	0.93
11	MainFrame	171	171	171	171	171	171	171	171	187	187	16	1.05
12	NFA	470	514	514	514	507	507	507	540	557	557	101	3.12
13	Out	149	154	154	167	167	167	167	167	189	189	40	2.48
14	RegExps	83	83	83	83	83	83	83	97	97	97	14	1.03
15	ScannerException	30	30	30	30	30	30	30	30	30	30	0	0

S.No.	Class Name	v1.3.0	v1.3.1	v1.3.2	v1.3.3	v1.3.4	v1.3.5	v1.4pre1	v1.4pre3	v1.4pre4	v1.4pre5	Change	IPC
16	SemCheck	144	163	163	163	163	163	163	163	163	163	19	1.83
17	Skeleton	311	311	311	313	313	313	313	313	91	91	224	3.5
18	StatePairList	41	41	41	41	41	41	41	41	40	40	1	0.97
19	StateSet	169	169	169	169	169	169	169	196	196	196	27	1.91
20	StateSetEnumerator	58	58	58	58	58	58	58	70	70	70	12	1.23
21	sym	48	48	48	48	48	48	48	50	50	50	2	0.23

A8. Experiments with jfreechart

S.No.	Class Name	v1.0.1	v1.0.2	v1.0.3	v1.0.4	v1.0.5	v1.0.6	v1.0.7	v1.0.8	v1.0.9	v1.0.10	v1.0.11	v1.0.12	Change	IPC
1	AbstractBlock	235	235	237	237	263	263	263	263	263	263	263	263	28	1.11
2	AbstractDataset	77	77	72	72	72	72	72	72	72	72	69	69	8	1.14
3	AbstractIntervalXYDataset	36	36	36	36	36	36	36	36	36	36	36	36	0	0.97
4	AbstractRenderer	1307	1277	1277	1277	1278	1351	1351	1351	1364	1364	1490	1490	243	6.18
5	AbstractSeriesDataset	25	25	25	25	25	25	25	25	25	25	23	23	2	1.84
6	AbstractXYAnnotation	65	65	65	64	74	74	74	74	74	74	74	74	11	1.23
7	AbstractXYDataset	25	25	25	25	25	25	25	25	25	25	25	25	0	0
8	AbstractXYItemRenderer	796	788	800	819	946	949	958	958	958	969	974	974	194	1.56
9	AreaRenderer	158	158	163	169	169	173	173	173	173	173	183	183	25	2
10	AreaRendererEndType	48	48	48	48	48	48	48	48	48	48	45	45	3	1.02
11	ArrowNeedle	71	71	71	71	71	71	71	77	77	77	77	77	6	0.76
12	Axis	561	561	544	544	544	544	544	544	544	549	546	587	66	1.86
13	AxisCollection	47	47	47	47	47	47	47	47	47	47	47	47	0	1.15
14	AxisLocation	75	75	75	72	74	74	74	74	74	74	74	74	5	1.04
15	AxisSpace	186	186	186	186	186	186	186	186	186	186	186	186	0	0.92
16	AxisState	59	59	59	59	59	59	59	59	59	59	59	59	0	0
17	BarChartDemo1	102	102	96	96	96	96	96	96	96	96	96	96	6	0.82
18	BarRenderer	568	551	551	554	554	560	560	560	560	579	672	675	141	4.64
19	BarRenderer3D	456	456	422	447	447	450	450	450	450	449	449	449	63	0.91
20	BlockBorder	101	101	101	101	101	101	101	101	101	101	101	101	0	0
21	BlockContainer	122	122	122	122	122	122	122	122	122	122	120	120	2	1.28
22	BlockParams	29	29	29	29	29	29	29	29	29	29	29	29	0	0
23	BorderArrangement	414	414	414	414	414	414	414	414	369	369	369	369	45	1.14
24	BoxAndWhiskerCalculator	112	112	121	121	121	121	121	121	121	121	121	121	9	0.73
25	BoxAndWhiskerItem	107	107	111	109	109	109	117	117	117	117	117	117	14	1.12
26	BoxAndWhiskerRenderer	563	563	499	503	503	510	510	510	516	497	503	506	109	5.14
27	CandlestickRenderer	393	393	404	404	428	428	445	445	445	445	445	445	52	1.61
28	CategoryAnchor	44	44	44	44	44	44	44	44	44	44	44	44	0	0.54
29	CategoryAxis	644	644	654	654	644	644	665	665	665	664	677	678	56	0.93
30	CategoryAxis3D	101	101	96	95	95	95	95	95	95	95	95	95	6	0.98
31	CategoryDatasetHandler	46	46	46	46	46	46	46	46	46	46	46	46	0	0.96
32	CategoryItemEntity	69	69	69	69	69	116	116	116	116	116	116	116	47	2.27
33	CategoryItemRenderer	204	204	204	204	204	203	203	203	203	203	188	188	16	0.87

S.No.	Class Name	v1.0.1	v1.0.2	v1.0.3	v1.0.4	v1.0.5	v1.0.6	v1.0.7	v1.0.8	v1.0.9	v1.0.10	v1.0.11	v1.0.12	Change	IPC
34	CategoryItemRendererState	24	24	24	24	24	24	24	24	24	24	32	32	8	1.32
35	CategoryLabelPosition	120	120	120	120	120	111	111	111	111	111	111	111	9	0.89
36	CategoryLabelPositions	264	264	264	264	264	264	264	264	264	264	264	264	0	1.17
37	CategoryLabelWidthType	44	44	44	44	44	44	44	44	44	44	44	44	0	0.92
38	CategoryLineAnnotation	185	185	185	185	195	195	196	196	196	197	197	197	12	0.37
39	CategoryMarker	52	52	61	61	61	61	61	61	61	61	61	61	9	1.98
40	CategoryPlot	1685	1701	1751	1756	1787	1791	1882	1882	1882	1951	2146	2201	516	2.82
41	CategorySeriesHandler	56	56	56	56	56	56	56	56	56	56	56	56	0	0.92
42	CategoryStepRenderer	123	123	123	126	170	171	171	171	171	170	176	176	55	2.63
43	CategoryTableXYDataset	114	114	114	114	114	114	114	114	114	124	124	124	10	2.88
44	CategoryTextAnnotation	122	122	111	111	119	119	119	119	119	120	120	120	20	1.35
45	CategoryTick	55	55	55	55	55	55	55	55	55	55	55	55	0	0.83
46	CategoryToPieDataset	92	92	149	149	149	149	149	149	149	149	149	149	57	1.03
47	CenterArrangement	141	141	137	137	137	137	137	137	137	137	137	137	4	1.45
48	ChartChangeEvent	31	31	31	31	31	31	31	31	31	31	31	31	0	0.64
49	ChartChangeEventType	47	47	47	47	47	47	47	47	47	47	47	47	0	0.51
50	ChartDeleter	34	34	34	35	35	35	35	35	35	35	35	35	1	1.76
51	ChartEntity	178	178	178	174	174	174	181	181	184	184	184	184	14	0.99
52	ChartProgressEvent	33	33	33	33	33	33	33	33	33	33	33	33	0	0
53	ClusteredXYBarRenderer	153	153	148	154	154	180	180	180	180	180	195	195	52	2.06
54	ColorBar	243	243	243	245	245	228	228	228	228	228	228	228	19	0.85
55	ColorBlock	21	21	21	21	54	54	55	55	55	63	63	63	42	1.13
56	ColorPalette	216	216	216	217	217	217	217	217	217	217	217	217	1	0.52
57	ColumnArrangement	207	207	206	206	206	206	206	206	206	206	206	206	1	1.91
58	CombinedDataset	235	235	235	235	235	235	235	235	235	235	233	233	2	1.34
59	CombinedDomainCategoryPlot	322	322	316	316	316	336	336	336	336	345	345	345	35	0.62
60	CombinedDomainXYPlot	321	321	321	321	321	341	341	341	341	347	340	340	33	1.83
61	CombinedRangeCategoryPlot	298	298	292	292	292	292	292	292	292	292	292	292	6	1.72
62	CombinedRangeXYPlot	309	309	304	304	304	325	326	326	326	330	326	326	35	0.62
63	CompassFormat	36	36	36	36	36	36	36	36	36	36	36	36	0	0
64	CompassPlot	428	428	428	428	444	444	444	444	444	444	444	445	17	0.55
65	CompositeTitle	65	65	65	65	65	65	65	65	63	63	91	91	30	1.71
66	ContourEntity	36	36	36	36	36	38	38	38	38	38	38	38	2	0.48
67	ContourPlot	938	938	938	893	893	893	893	893	893	893	890	892	50	2.09
68	CrosshairState	76	76	94	117	117	117	117	117	117	117	124	124	48	1.32
69	CSV	96	96	96	96	96	96	96	96	96	96	96	96	0	0
70	CustomPieURLGenerator	92	92	92	93	93	93	93	93	93	93	91	91	3	0.37
71	CustomXYToolTipGenerator	75	75	75	75	75	75	75	75	75	75	73	73	2	1.53
72	CustomXYURLGenerator	78	78	78	78	78	78	78	78	78	86	86	86	8	1.91
73	CyclicNumberAxis	675	675	675	675	675	675	676	676	676	676	672	672	5	1.75
74	CyclicXYItemRenderer	264	264	252	252	252	252	252	252	252	252	252	252	12	1.38
75	DatasetGroup	35	35	35	35	35	35	35	35	35	35	35	35	0	0.81
76	DatasetReader	59	59	59	59	59	59	59	59	59	59	59	59	0	0

S.No.	Class Name	v1.0.1	v1.0.2	v1.0.3	v1.0.4	v1.0.5	v1.0.6	v1.0.7	v1.0.8	v1.0.9	v1.0.10	v1.0.11	v1.0.12	Change	IPC
77	DatasetRenderingOrder	39	39	39	39	39	39	39	42	42	42	42	42	3	1.86
78	DatasetUtilities	876	876	876	876	887	887	885	885	885	916	916	939	67	1.25
79	DataUtilities	69	69	69	69	69	69	69	69	69	69	69	69	0	0
80	DateAxis	1014	1014	945	955	955	986	986	986	986	986	1004	1067	191	6.69
81	DateRange	37	37	37	37	37	37	37	37	37	37	43	43	6	0.41
82	DateTick	32	32	32	32	32	32	32	32	32	32	32	37	5	0.23
83	DateTickMarkPosition	44	44	44	44	44	44	44	44	44	44	44	44	0	0.78
84	DateTickUnit	147	147	147	147	153	166	166	166	166	160	160	160	25	0.36
85	DateTitle	48	48	48	39	39	39	39	39	39	39	39	39	9	1.76
86	Day	153	153	168	168	168	168	168	168	168	168	168	168	15	1.88
87	DefaultAxisEditor	220	220	220	220	220	220	220	220	220	220	220	222	2	0.33
88	DefaultCategoryDataset	132	132	132	132	137	137	137	137	137	138	138	138	6	0.56
89	DefaultChartEditor	137	137	137	137	137	137	137	137	137	137	137	139	2	1.92
90	DefaultColorBarEditor	111	111	111	111	111	111	111	111	111	111	111	113	2	1.67
91	DefaultContourDataset	218	218	218	219	219	219	219	219	219	219	219	219	1	0.31
92	DefaultDrawingSupplier	258	258	246	246	246	268	268	268	268	268	268	268	34	2.64
93	DefaultHighLowDataset	105	105	105	141	141	141	141	141	141	142	142	142	37	2.11
94	DefaultKeyedValue	50	50	50	50	50	53	53	53	53	56	55	55	7	0.48
95	DefaultKeyedValueDataset	74	74	74	74	74	74	74	74	74	74	73	73	1	0.36
96	DefaultKeyedValues	150	150	154	154	154	176	202	200	200	200	199	199	55	0.57
97	DefaultKeyedValues2D	234	234	234	239	239	242	242	260	260	260	259	259	27	2.81
98	DefaultNumberAxisEditor	196	196	196	196	196	196	196	196	196	196	196	198	2	0.56
99	DefaultOHLCDataset	104	104	104	104	104	104	104	104	104	111	111	111	7	1.8
100	DefaultPieDataset	108	108	119	119	119	126	126	126	126	126	124	124	20	0.88
101	DefaultPlotEditor	413	413	413	413	413	413	415	415	415	415	411	413	8	1.39
102	DefaultTableXYDataset	275	275	275	275	275	275	275	275	275	292	292	292	17	1.75
103	DefaultTitleEditor	158	158	158	158	158	158	158	158	158	158	158	160	2	1.12
104	DefaultValueDataset	43	43	43	41	41	41	41	41	41	41	39	39	4	0.96
105	DefaultWindDataset	126	126	176	176	176	176	176	176	176	177	177	177	51	2.17
106	DisplayChart	59	59	57	57	57	57	57	57	57	57	57	57	2	1.27
107	DomainOrder	46	46	46	46	46	46	46	46	46	46	46	46	0	0.51
108	DrawableLegendItem	68	68	68	68	68	68	68	68	68	68	68	68	0	0
109	DynamicTimeSeriesCollection	396	396	396	396	396	396	396	396	396	396	396	396	0	0
110	EmptyBlock	22	22	22	22	22	22	22	22	22	30	30	30	8	1.91
111	EncoderUtil	54	54	54	54	54	54	54	54	54	54	54	54	0	0
112	ExtendedCategoryAxis	62	62	62	62	105	105	106	106	106	106	106	106	44	2.54
113	FixedMillisecond	96	96	98	98	98	98	98	98	98	98	98	98	2	1.68
114	FlowArrangement	233	233	232	232	232	232	232	232	232	232	232	232	1	1.74
115	GanttRenderer	362	363	363	363	353	353	353	353	353	353	360	360	18	2.61
116	GridArrangement	145	145	145	145	145	145	145	145	145	145	145	242	97	2.21
117	GroupedStackedBarRenderer	251	251	251	251	251	241	241	241	218	218	222	225	40	2.71
118	HighLowItemLabelGenerator	98	98	98	98	98	98	96	96	96	100	100	100	6	1.49
119	HighLowRenderer	247	247	241	241	241	241	241	241	241	260	245	247	42	2.12

S.No.	Class Name	v1.0.1	v1.0.2	v1.0.3	v1.0.4	v1.0.5	v1.0.6	v1.0.7	v1.0.8	v1.0.9	v1.0.10	v1.0.11	v1.0.12	Change	IPC
120	HistogramBin	52	52	51	51	51	51	51	51	51	51	51	51	1	1.25
121	HistogramDataset	196	196	198	198	198	198	198	198	198	203	203	203	7	1.21
122	HistogramType	50	50	50	50	50	50	50	50	50	50	50	50	0	0
123	Hour	162	162	176	176	176	176	176	176	176	176	180	180	18	2.6
124	ImageEncoderFactory	62	62	62	62	62	62	62	62	62	62	62	62	0	0
125	ImageMapUtilities	82	82	82	73	73	73	73	73	106	106	106	106	42	0.89
126	ImageTitle	139	139	139	139	139	139	139	139	139	160	160	160	21	2.5
127	IntervalBarRenderer	167	167	167	158	158	157	157	157	157	135	129	134	43	2.27
128	IntervalXYDelegate	169	169	169	169	169	169	169	169	169	169	168	168	1	0.97
129	ItemHandler	60	60	60	60	60	60	60	60	60	60	60	60	0	0
130	ItemLabelAnchor	155	155	155	155	155	155	155	155	155	155	155	155	0	1.04
131	ItemLabelPosition	76	76	76	72	72	72	72	72	72	72	72	72	4	1.08
132	JDBCCategoryDataset	139	139	139	139	139	139	140	140	140	140	140	140	1	1.22
133	JDBCPieDataset	106	106	106	106	106	106	107	107	107	107	107	107	1	0.45
134	JDBCXYDataset	270	270	270	270	270	270	270	270	270	270	268	268	2	0.38
135	JFreeChart	877	834	834	837	850	874	875	875	881	886	890	892	101	3.83
136	KeyHandler	50	50	50	50	50	50	50	50	50	50	50	50	0	1.13
137	KeypointPNGEncoderAdapter	41	41	41	41	41	41	41	41	41	41	41	41	0	0
138	NonGridContourDataset	108	108	108	110	110	110	110	110	110	110	110	110	2	0.35
139	PaletteChooserPanel	21	21	21	23	23	23	23	23	23	23	23	23	2	1.48
140	PaletteSample	64	64	64	66	66	66	66	66	66	66	66	66	2	0.69
141	PieChartDemo1	53	53	53	53	53	53	52	52	52	52	49	49	4	0.84
142	PieDatasetHandler	43	43	43	43	43	43	43	43	43	43	43	43	0	0
143	RootHandler	38	38	38	38	38	38	38	38	38	38	38	38	0	0
144	ServletUtilities	199	183	184	184	184	184	184	184	184	184	184	184	17	2.16
145	SunJPEGEncoderAdapter	37	37	56	56	56	56	56	56	56	56	56	56	19	0.86
146	SunPNGEncoderAdapter	33	33	33	33	33	33	33	33	33	33	33	33	0	0
147	TimeSeriesChartDemo1	112	112	110	110	110	110	110	110	110	110	111	111	3	0.62
148	ValueHandler	57	57	57	57	57	57	57	57	57	57	57	57	0	0

A9. Experiments with jUnit

Following table tabulates LOC measure, IPC and Change metric of each class in different releases of jUnit from versions 4.5 through version 4.10.

S.No.	Class Name	v4.5	v4.6	v4.7	v4.8	v4.8.1	v4.9	v4.10	Change	IPC
1	ActiveTestSuite	52	52	52	52	52	52	52	0	0
2	AllMembersSupplier	100	99	99	99	99	99	100	2	0.49
3	AnnotatedBuilder	36	36	36	36	36	36	36	0	0
4	ArrayComparisonFailure	36	36	36	36	36	36	36	0	0.63
5	Assert	146	146	146	147	147	151	151	5	0.37
6	Assert	238	259	223	223	223	225	225	59	0.49

S.No.	Class Name	v4.5	v4.6	v4.7	v4.8	v4.8.1	v4.9	v4.10	Change	IPC
7	Assignments	99	99	101	101	101	101	101	2	0.24
8	Assume	23	23	23	23	23	23	23	0	0
9	AssumptionViolatedException	32	32	32	32	32	32	32	0	0
10	BaseTestRunner	236	236	235	235	235	235	235	1	0.35
11	BlockJUnit4ClassRunner	166	163	213	213	213	199	207	75	0.51
12	ClassRoadie	65	65	65	65	65	65	65	0	0
13	CombinableMatcher	26	26	26	26	26	26	26	0	0
14	ComparisonCompactor	60	60	60	60	60	60	60	0	0.55
15	ComparisonFailure	22	22	22	22	22	22	22	0	0
16	ComparisonFailure	81	81	81	81	81	81	81	0	0
17	Description	82	111	114	114	114	113	115	35	0.92
18	Each	20	20	20	20	20	20	20	0	0
19	EachTestNotifier	34	34	34	34	34	38	38	4	0.36
20	ErrorReportingRunner	51	50	50	50	50	50	50	1	0.23
21	ExpectException	28	28	28	28	28	31	31	3	0.22
22	FailOnTimeout	50	50	36	36	36	36	54	32	0.88
23	Failure	37	37	37	37	37	37	39	2	0.17
24	FailureList	23	23	23	23	23	23	23	0	0
25	Filter	23	40	40	40	40	64	64	41	1.04
26	FilterRequest	26	26	26	26	26	26	26	0	0
27	FrameworkMethod	84	84	80	80	80	80	85	9	0.38
28	InitializationError	20	20	20	20	20	20	20	0	0.2
29	IsCollectionContaining	54	54	54	54	54	54	54	0	0
30	JUnit38ClassRunner	108	142	134	134	134	134	134	42	0.77
31	JUnit4ClassRunner	117	117	117	117	117	117	117	0	0
32	JUnit4TestAdapter	66	66	66	66	66	66	66	0	0.64
33	JUnit4TestAdapterCache	67	67	67	67	67	67	67	0	0
34	JUnit4TestCaseFacade	23	23	23	23	23	23	23	0	0.36
35	JUnitCore	81	87	87	87	87	87	87	6	0.33
36	JUnitMatchers	32	32	32	32	32	32	32	0	0.45
37	MethodRoadie	138	138	138	138	138	138	138	0	0.55
38	MethodValidator	72	72	72	72	72	72	72	0	0
39	Parameterized	93	93	94	94	94	94	99	6	0.16
40	ParameterizedAssertionError	38	38	38	38	38	38	38	0	0
41	ParameterSignature	71	71	71	71	71	71	71	0	0
42	ParentRunner	155	155	171	171	171	204	207	52	1.13
43	PotentialAssignment	25	25	25	25	25	25	25	0	0.11
44	PrintableResult	32	32	32	32	32	33	33	1	0.3
45	RepeatedTest	28	28	28	28	28	28	28	0	0
46	Request	66	57	56	56	56	56	56	10	0.63
47	Result	63	59	59	61	61	61	63	8	0.47
48	ResultMatchers	41	41	41	41	41	41	41	0	0
49	ResultPrinter	92	92	92	92	92	92	92	0	0.45

S.No.	Class Name	v4.5	v4.6	v4.7	v4.8	v4.8.1	v4.9	v4.10	Change	IPC
50	RunAfters	38	38	34	34	34	34	33	5	0.49
51	RunBefores	20	20	20	20	20	20	20	0	0.8
52	RunListener	20	20	20	20	20	20	20	0	0
53	RunnerBuilder	44	44	44	44	44	48	48	4	0.2
54	RunNotifier	93	93	93	97	97	97	97	4	0.34
55	Sorter	22	22	22	22	22	22	22	0	0
56	SortingRequest	20	20	20	20	20	20	20	0	0
57	StringContains	20	20	20	20	20	20	20	0	0
58	SubstringMatcher	20	20	20	20	20	20	20	0	0.57
59	Suite	54	55	63	63	63	63	63	9	0.39
60	SuiteMethod	24	24	24	24	24	24	24	0	0.42
61	SuiteMethodBuilder	20	20	20	20	20	20	20	0	0.5
62	TestCase	81	81	81	81	81	81	81	0	0
63	TestClass	81	81	81	81	81	81	81	0	0
64	TestClass	82	82	85	85	85	104	108	26	0.48
65	TestDecorator	26	26	26	26	26	26	26	0	0
66	TestFailure	36	36	36	36	36	36	36	0	0.34
67	TestMethod	51	51	51	51	51	51	51	0	0
68	TestResult	100	100	100	100	100	100	100	0	0
69	TestRunner	124	124	124	124	124	124	124	0	0
70	TestSetup	24	24	24	24	24	24	24	0	0
71	TestSuite	164	164	164	164	164	173	173	9	0.16
72	TextListener	73	73	73	73	73	73	73	0	0
73	Theories	155	158	159	159	159	165	165	10	1.18
74	TypeSafeMatcher	34	34	34	34	34	34	34	0	0

APPENDIX B *Source Code*

B1. Source code of Triangle classification program used for case study 1 in chapter 5.

```
// Jeff Offutt -- Java version Feb 2003


// Classify triangles
import java.io.*;
class Side
{
        int length;
        Side()
        {
                length = 0;
        }
        void setSide(int x)
        {
                length = x;
        }
        int getSide()
        {
                return length;
        }
}

class TriangleTest
{
  private static String[] triTypes = { "", "scalene",
"isosceles", "equilateral", "not a valid triangle"};
  private static String instructions = "This is the ancient
" +
  "TriangleTest program.\nEnter three integers that
represent the " +
  "lengths of the sides of a triangle.\nThe triangle will be
" +
  "categorized as either scalene, isosceles, equilateral\n"
+
  "or invalid.\n";

public static void main (String[] argv)
{ // Driver program for TriangleTest
  Side A = new Side();
  Side B = new Side();
  Side C = new Side();

  int T;

  System.out.println (instructions);
  System.out.println ("Enter side 1: ");
  A.setSide(getN());
  System.out.println ("Enter side 2: ");
  B.setSide(getN());
  System.out.println ("Enter side 3: ");
  C.setSide(getN());
  T = Triang (A, B, C);

  System.out.println ("Result is: " + triTypes[T]);
}

// ===================================
// The main triangle classification method
static int Triang (Side Side1, Side Side2, Side Side3)
{
  int triOut;

  // triOut is output from the routine:
  //  Triang = 1 if triangle is scalene
  //  Triang = 2 if triangle is isosceles
  //  Triang = 3 if triangle is equilateral
  //  Triang = 4 if not a triangle

  // After a quick confirmation that it's a legal
  // triangle, detect any sides of equal length
  if (Side1.getSide() <= 0 || Side2.getSide() <= 0 ||
Side3.getSide() <= 0)
  {
   triOut = 4;
            return (triOut);
  }

  triOut = 0;
  if (Side1.getSide() == Side2.getSide())
   triOut = triOut + 1;
  if (Side1.getSide() == Side3.getSide())
   triOut = triOut + 2;
  if (Side2.getSide() == Side3.getSide())
   triOut = triOut + 3;
  if (triOut == 0)
  { // Confirm it's a legal triangle before declaring
   // it to be scalene

   if (Side1.getSide() + Side2.getSide() <=
Side3.getSide()
                    || Side2.getSide() + Side3.getSide()
<= Side1.getSide()
                    || Side1.getSide() + Side3.getSide()
<= Side2.getSide())
    triOut = 4;
   else
```

```
    triOut = 1;
   return (triOut);
   }

   /* Confirm it's a legal triangle before declaring */
   /* it to be isosceles or equilateral */

   if (triOut > 3)
    triOut = 3;
   else if (triOut == 1 && Side1.getSide() +
Side2.getSide() > Side3.getSide())
    triOut = 2;
   else if (triOut == 2 && Side1.getSide() +
Side3.getSide() > Side2.getSide())
    triOut = 2;
   else if (triOut == 3 && Side2.getSide() +
Side3.getSide() > Side1.getSide())
    triOut = 2;
   else
    triOut = 4;
   return (triOut);
} // end Triang

// ===================================
// Read (or choose) an integer
private static int getN ()
{
  int inputInt = 1;
  BufferedReader in = new BufferedReader (new
InputStreamReader (System.in));
  String inStr;

  try
  {
   inStr = in.readLine ();
   inputInt = Integer.parseInt(inStr);
  }
  catch (IOException e)
  { // JDK requires the IOException to be caught.
   System.out.println ("Could not read input, choosing
1.");
  }
  catch (NumberFormatException e)
  {
   System.out.println ("Entry must be a number,
choosing 1.");
  }

   return (inputInt);
  } // end getN

} // end TriangleTest class
```

B2. Following is the program code used for test case generation in case study 1 in chapter 6.

```
package genut;
import genut.encoding.tree.*;
import java.util.*;

public class GARunner
{
    private static int num_cases = 0;
    private static int maxIteration = 20;
    private static int populationSize = 40;
    private static int iterationCounter = 0;
    private static int covered_nodes;
    private static int num_visit;
    private static int cur_node;
    private static int genNo;
    private static float sum_wt;
    private static double galat = 0.001;

    private static boolean stopFactor = false;
    private static double crossover_rate= 0.6;
    private static double mutation_rate= 0.1;
    private static RecordFile recFile = new RecordFile();
    private static ArrayList<TreeChromosome>
population = new ArrayList();
    private static ReplacementScheme
replacementScheme;
    private static SelectionScheme selection;
    private static Random randGenerator = new
Random();

    public static TreeChromosome[]
runIteration(TreeChromosome[] currentGeneration,
TreeChromosome t1)
    {
        TreeChromosome[] selectedPopulation =
selection.select(currentGeneration);
        List<TreeChromosome> parentsList = new
ArrayList<>();
        Triangle[] tri = new Triangle[populationSize];
        Node[] nodes = new Node[22];
        Record record;
        setReplacementScheme(new
ReplacementScheme());
        int T;
        double coverage;
        double denom;
        initializeNodes(nodes);
        for(int i=0;i<populationSize;i++)
        {
            t1 = population.get(i);
```

```
        tri[i] = new
Triangle(t1.getElement(0).intval,t1.getElement(1).intval
,t1.getElement(2).intval);
        T = Triang (tri[i].gets1(), tri[i].gets2(),
tri[i].gets3(),nodes);
        coverage = ((float)covered_nodes/22)*100;
        t1.setCoverage(coverage);
        if(T==4)
        {
           denom =
((double)sum_wt/num_visit)*Node.init_wt;
           t1.setFeasible(false);

t1.setFitness(compute_num(cur_node,nodes)/denom);
        }
        else
        {
           t1.setFeasible(true);
           t1.setFitness((double)sum_wt/num_visit);
        }
        record = new Record(genNo, i, coverage,
t1.getFeasible());
        recFile.addRecord(record);
     }

     for (TreeChromosome parentCandidate:
selectedPopulation)
     {
        if ( randGenerator.nextDouble() <=
crossover_rate )
        {
             parentsList.add(parentCandidate);
        }
     }

     TreeChromosome[] offsprings = new
TreeChromosome[parentsList.size()];
     TreeChromosome[] currentOffsprings;
     int offspringCounter = 0;
     int crossoverCounter;
     int stopCrossover;
     int parentsSize = parentsList.size();

     if (parentsSize % 2 == 0)
     {
        stopCrossover = parentsSize;
     }
     else
     {
        stopCrossover = parentsSize - 1;
     }

     for(crossoverCounter = 0; crossoverCounter <
stopCrossover; crossoverCounter += 2)
     {
        currentOffsprings =
parentsList.get(crossoverCounter).crossover(parentsList
.get(crossoverCounter+1));

        for (TreeChromosome currentOffspring:
currentOffsprings)
        {
             offsprings[offspringCounter++] =
currentOffspring;
        }
     }

     if (parentsSize %2 != 0)
     {
        currentOffsprings =
parentsList.get(crossoverCounter).crossover(parentsList
.get(crossoverCounter));
        offsprings[offspringCounter] =
currentOffsprings[0];
     }

     if (mutation_rate > 0.00)
     {
        for(TreeChromosome c: offsprings)
        {
           c.mutate(mutation_rate);
        }
     }

     iterationCounter++;
     stopFactor = checkStopFactor(currentGeneration,
offsprings);
     currentGeneration =
replacementScheme.replace(currentGeneration,
offsprings);

     return currentGeneration;
  }

  public static void
runGeneticAlgorithm(ArrayList<TreeChromosome>
initPopulation, TreeChromosome t1)
  {
     genNo = 0;
     TreeChromosome[] chromo =
(TreeChromosome[])initPopulation.toArray();
     do
     {
        genNo++;
        chromo = runIteration(chromo, t1);
     }
     while(!getStopFactor());
  }

  private static void initializeNodes(Node[] nodes)
  {
```

```
      for(int i=0;i<22;i++)
      {
         nodes[i] = new Node(i+1);
      }
   }

   public static void
setReplacementScheme(ReplacementScheme
repScheme)
   {
      if (replacementScheme != null)
      {
            replacementScheme = repScheme;
      }
   }

   public static boolean getStopFactor()
   {
      return stopFactor;
   }

   public static boolean
checkStopFactor(TreeChromosome[] parents,
TreeChromosome[] offsprings)
   {
      if (iterationCounter >= maxIteration)
      {
         return true;
      }
      else
      {
         Comparator comparator = new
FitnessComparator();
         Arrays.sort(parents, comparator);
         Arrays.sort(offsprings, comparator);

         if ( Math.abs( parents[parents.length-1].fitness()
- offsprings[offsprings.length-1].fitness() ) <= galat )
         {
               return true;
         }
         else
         {
               return false;
         }
      }
   }

   public static void populate(int
population_size,TreeChromosome tree)
   {
      ArrayList<TreeChromosome> lst = tree.getList();
      int i;
      NodeGene z1;
      for (i=0;i<population_size;i++)
      {
         while((z1=tree.getElement(i))!=null)
         {
            traverse(z1);
            i++;
         }
      }
   }
   static void crossover(ArrayList<TreeChromosome>
pop)
   {
      int val;
      int size;
      ArrayList<TreeChromosome> temp = new
ArrayList();
      FitnessComparator fitComp = new
FitnessComparator();
      TreeFitnessFunction treeFit = new
TreeFitnessFunction();
      TreeChromosome[] offSprings;
      TreeCrossover crossObj = new
TreeCrossover(treeFit);
      TreeChromosome t1;
      TreeChromosome t2;
      while(pop.isEmpty())
      {
         t1 = pop.remove(0);
         t2 = pop.remove(1);
         if((t1!=null)&&(t2!=null))
         {
            val = fitComp.compare(t1, t2);
            if(val<0)
            {
               temp.add(t2);
               temp.add(t1);
            }
            else
            {
               temp.add(t1);
               temp.add(t2);
            }
         }
         else
         {
            temp.add(t1);
         }
      }
      size=temp.size()/2;
      for(int i=0;i<size;i=i+2)
      {
         offSprings = crossObj.crossover(temp.get(i),
temp.get(i+1));
         pop.add(temp.get(i));
         pop.add(temp.get(i+1));
         pop.add(offSprings[0]);
         pop.add(offSprings[1]);
      }
```

```
    }
    static void mutation(ArrayList<TreeChromosome>
pop)
    {
        int i=0;
        int j;
        TreeChromosome tr1;
        NodeGene elm;
        NodeGene elm1;
        MethodNodeGene elm2;
        while(pop.get(i)!=null)
        {
            tr1 = pop.get(i);
            j=0;
            while(tr1.getElement(j)!=null)
            {
                elm = tr1.getElement(i);
                switch (elm.type) {
                    case "Constr":
                        elm1 = (NodeGene)elm;
                        ConstrNodeMutation.mutate(elm1);
                        break;
                    case "Method":
                        elm2 = (MethodNodeGene)elm;
                        MethodNodeMutation.mutate(elm2);
                        break;
                }
                j++;
            }
            i++;
        }
    }

    public static void traverse(NodeGene z)
    {
        int i = 0;
        switch (z.type) {
            case "int":
                z.intval=recFile.assgnintval();
                break;
            case "float":
                z.floatval=recFile.assgnfloatval();
                break;
            case "boolean":
                z.boolval=recFile.assgnboolval();
                break;
        }
    }

    static int Triang (Side Side1, Side Side2, Side Side3,
Node[] nodes)
    {
        int triOut;
        float hcf;
        sum_wt=(float)0.0;
        num_visit=0;
        covered_nodes=0;
        num_cases++;
        // triOut is output from the routine:
        //   Triang = 1 if triangle is scalene
        //   Triang = 2 if triangle is isosceles
        //   Triang = 3 if triangle is equilateral
        //   Triang = 4 if not a triangle

       call_visit(0,nodes);

        if (Side1.getSide() <= 0 || Side2.getSide() <= 0 ||
Side3.getSide() <= 0)
        {
            call_visit(1,nodes);
            triOut = 4;
            return (triOut);
        }

        triOut = 0;
             call_visit(2,nodes);

             call_visit(3,nodes);

        if (Side1.getSide() == Side2.getSide())
        {
            triOut = triOut + 1;
            call_visit(4,nodes);
        }
             call_visit(5,nodes);

        if (Side1.getSide() == Side3.getSide())
        {
            triOut = triOut + 2;
            call_visit(6,nodes);
        }
             call_visit(7,nodes);
        if (Side2.getSide() == Side3.getSide())
        {
            triOut = triOut + 3;
            call_visit(8,nodes);
        }
             call_visit(9,nodes);
        if (triOut == 0)
        {
            call_visit(10,nodes);
            if (Side1.getSide() + Side2.getSide() <=
Side3.getSide()
                        || Side2.getSide() + Side3.getSide()
<= Side1.getSide()
                        || Side1.getSide() + Side3.getSide()
<= Side2.getSide())
            {
                triOut = 4;
                        call_visit(11,nodes);
            }
            else
```

```
        {
           triOut = 1;
                    call_visit(12,nodes);
        }
        return (triOut);
      }
          call_visit(13,nodes);
      if (triOut > 3)
      {
        triOut = 3;
        call_visit(14,nodes);
        call_visit(15,nodes);
      }
      else if (triOut == 1 && Side1.getSide() +
Side2.getSide() > Side3.getSide())
      {
        triOut = 2;
        call_visit(16,nodes);
        call_visit(17,nodes);
      }
      else if (triOut == 2 && Side1.getSide() +
Side3.getSide() > Side2.getSide())
      {
        triOut = 2;
        call_visit(18,nodes);
        call_visit(19,nodes);
      }

      else if (triOut == 3 && Side2.getSide() +
Side3.getSide() > Side1.getSide())
      {
        triOut = 2;
        call_visit(20,nodes);
      }
      else
      {
        triOut = 4;
        call_visit(21,nodes);
      }
      return (triOut);
   }

   static void call_visit(int n, Node[] nodes)
   {
      float hcf;

System.out.println("Node:"+nodes[n].getNumber());
      nodes[n].incVisit();
      System.out.println("visited:"+nodes[n].getVisit());
      covered_nodes++;
      hcf = (float)(num_cases -
nodes[n].getVisit())/num_cases;
      nodes[n].sethcf(hcf);
      compute_pf(n, nodes);
      nodes[n].wt_reval();
      sum_wt+=nodes[n].getWeight();
      num_visit++;
      cur_node=n;
   }

   static float compute_num(int n, Node[] nodes)
   {
      float den=(float)0.0;
      float temp=(float)0.0;
      int i;
      if(nodes[n].getNumber()==1)
      {

for(i=1;i<22;i++)temp+=(float)nodes[n].getWeight();
         den=temp/21;
      }
      if(nodes[n].getNumber()==2)
      den=1;
      if(nodes[n].getNumber()==3)
      {
         for(i=3;i<22;i++)
         temp+=(float)nodes[n].getWeight();
         den=temp/19;
      }
      if(nodes[n].getNumber()==4)
      {
         for(i=4;i<22;i++)
         temp+=(float)nodes[n].getWeight();
         den=temp/18;
      }
      if(nodes[n].getNumber()==5)
      {
         for(i=5;i<22;i++)
         temp+=(float)nodes[n].getWeight();
         den=temp/17;
      }
      if(nodes[n].getNumber()==6)
      {
         for(i=6;i<22;i++)
         temp+=(float)nodes[n].getWeight();
         den=temp/16;
      }
      if(nodes[n].getNumber()==7)
      {
         for(i=7;i<22;i++)
         temp+=(float)nodes[n].getWeight();
         den=temp/15;
      }
      if(nodes[n].getNumber()==8)
      {
         for(i=8;i<22;i++)
         temp+=(float)nodes[n].getWeight();
         den=temp/14;
      }
      if(nodes[n].getNumber()==9)
      {
         for(i=9;i<22;i++)
```

```
        temp+=(float)nodes[n].getWeight();
        den=temp/13;
    }
    if(nodes[n].getNumber()==10)
    {
        for(i=10;i<22;i++)
        temp+=(float)nodes[n].getWeight();
        den=temp/12;
    }
    if(nodes[n].getNumber()==11)
    {
        for(i=11;i<13;i++)
        temp+=(float)nodes[n].getWeight();
        den=temp/2;
    }
    if(nodes[n].getNumber()==12)
    den=1;
    if(nodes[n].getNumber()==13)
    den=1;
    if(nodes[n].getNumber()==14)
    {
        for(i=14;i<22;i++)
        temp+=(float)nodes[n].getWeight();
        den=temp/8;
    }
    if(nodes[n].getNumber()==15)
    den=1;
    if(nodes[n].getNumber()==16)
    {
        for(i=16;i<22;i++)
        temp+=(float)nodes[n].getWeight();
        den=temp/6;
    }
    if(nodes[n].getNumber()==17)
    den=1;
    if(nodes[n].getNumber()==18)
    {
        for(i=18;i<22;i++)
        temp+=(float)nodes[n].getWeight();
        den=temp/4;
    }
    if(nodes[n].getNumber()==19)
    den=1;
    if(nodes[n].getNumber()==20)
    {
        for(i=20;i<22;i++)
        temp+=(float)nodes[n].getWeight();
        den=temp/2;
    }
    if(nodes[n].getNumber()==21)
    den=1;
    if(nodes[n].getNumber()==22)
    den=1;
    return den;
}
```

```
    static void compute_pf(int n, Node[] nodes)
    {
        float ppf=(float)0.0;
        float temp=(float)0.0;
        int i;
        if(nodes[n].getNumber()==1)
        {
            for(i=1;i<22;i++)
            {

temp+=(float)nodes[n].getWeight()/Node.init_wt;
            }
            ppf=temp/21;
        }
        if(nodes[n].getNumber()==2)
        {
            ppf=1;
        }
        if(nodes[n].getNumber()==3)
        {
            for(i=3;i<22;i++)
            {

temp+=(float)nodes[n].getWeight()/Node.init_wt;
            }
            ppf=temp/19;
        }
        if(nodes[n].getNumber()==4)
        {
            for(i=4;i<22;i++)
            {

temp+=(float)nodes[n].getWeight()/Node.init_wt;
            }
            ppf=temp/18;
        }
        if(nodes[n].getNumber()==5)
        {
            for(i=5;i<22;i++)
            {

temp+=(float)nodes[n].getWeight()/Node.init_wt;
            }
            ppf=temp/17;
        }
        if(nodes[n].getNumber()==6)
        {
            for(i=6;i<22;i++)
            {

temp+=(float)nodes[n].getWeight()/Node.init_wt;
            }
            ppf=temp/16;
        }
        if(nodes[n].getNumber()==7)
        {
```

```
            for(i=7;i<22;i++)
            {

temp+=(float)nodes[n].getWeight()/Node.init_wt;
            }
            ppf=temp/15;
        }
        if(nodes[n].getNumber()==8)
        {
            for(i=8;i<22;i++)
            {

temp+=(float)nodes[n].getWeight()/Node.init_wt;
            }
            ppf=temp/14;
        }
        if(nodes[n].getNumber()==9)
        {
            for(i=9;i<22;i++)
            {

temp+=(float)nodes[n].getWeight()/Node.init_wt;
            }
            ppf=temp/13;
        }
        if(nodes[n].getNumber()==10)
        {
            for(i=10;i<22;i++)
            {

temp+=(float)nodes[n].getWeight()/Node.init_wt;
            }
            ppf=temp/12;
        }
        if(nodes[n].getNumber()==11)
        {
            for(i=11;i<13;i++)
            {

temp+=(float)nodes[n].getWeight()/Node.init_wt;
            }
            ppf=temp/2;
        }
        if(nodes[n].getNumber()==12)
        {
            ppf=1;
        }
        if(nodes[n].getNumber()==13)
        {
            ppf=1;
        }
        if(nodes[n].getNumber()==14)
        {
            for(i=14;i<22;i++)
            {

temp+=(float)nodes[n].getWeight()/Node.init_wt;
            }
            ppf=temp/8;
        }
        if(nodes[n].getNumber()==15)
        {
            ppf=1;
        }
        if(nodes[n].getNumber()==16)
        {
            for(i=16;i<22;i++)
            {

temp+=(float)nodes[n].getWeight()/Node.init_wt;
            }
            ppf=temp/6;
        }
        if(nodes[n].getNumber()==17)
        {
            ppf=1;
        }
        if(nodes[n].getNumber()==18)
        {
            for(i=18;i<22;i++)
            {

temp+=(float)nodes[n].getWeight()/Node.init_wt;
            }
            ppf=temp/4;
        }
        if(nodes[n].getNumber()==19)
        {
            ppf=1;
        }
        if(nodes[n].getNumber()==20)
        {
            for(i=20;i<22;i++)
            {

temp+=(float)nodes[n].getWeight()/Node.init_wt;
            }
            ppf=temp/2;
        }
        if(nodes[n].getNumber()==21)
        {
            ppf=1;
        }
        if(nodes[n].getNumber()==22)
        {
            ppf=1;
        }
        nodes[n].setpf(ppf);
    }

    public static void main (String[] args)
```

```
    {
        TreeChromosome tree = recFile.encodeTree();
        populate(populationSize,tree);//Initial Population
        runGeneticAlgorithm(population,tree);
        recFile.printRecords();
    }
}

//Class FitnessComparator

package genut;

import genut.encoding.tree.TreeChromosome;
import java.util.Comparator;

public class FitnessComparator implements
Comparator<TreeChromosome>
{
    public int compare(TreeChromosome a,
TreeChromosome b)
    {
        double fitnessA = a.fitness();
        double fitnessB = b.fitness();

        if (fitnessA < fitnessB)
        {
            return Integer.MIN_VALUE;
        }
        else if (fitnessA > fitnessB)
        {
            return Integer.MAX_VALUE;
        }
        else
        {
            return 0;
        }
    }
}

//Class Node
package genut;

public class Node
{
    public static int init_wt = 300;
    private static float weight_factor=(float)1.0;
    private static float alpha = (float)1.0;
    private float weight;
    private float hcf;
    private float pf;
    private float wf;
    private int visit;
    private int number; //node number

    Node(int num)
    {
        weight = init_wt;
        hcf = (float)0.0;
        pf = wf = (float) 0.0;
        visit = 0;
        number = num;
    }
    public void wt_reval()
    {
        weight=alpha*weight*weight_factor*pf*hcf;
    }
    public void setWeiht(int wt)
    {
        weight = wt;
    }
     public void incVisit()
    {
        visit++;
    }
    public void setNumber(int num)
    {
        number = num;
    }
    public float getWeight()
    {
        return weight;
    }
    public int getVisit()
    {
        return visit;
    }
    public int getNumber()
    {
        return number;
    }
    public void sethcf(float val)
    {
        hcf = val;
    }
    public void setpf(float val)
    {
        pf = val;
    }
    public float gethcf()
    {
        return hcf;
    }
}

//Class Record
package genut;

class Record
{
    private int generationNo;
    private int individualNo;
    private double coverage;
```

```
    private boolean ifFeasible;

    public Record(int generationNo, int individualNo,
double coverage, boolean ifFeasible)
    {
        this.generationNo = generationNo;
        this.individualNo = individualNo;
        this.coverage = coverage;
        this.ifFeasible = ifFeasible;
    }

    public double getCoverage() {
        return coverage;
    }

    public void setCoverage(double coverage) {
        this.coverage = coverage;
    }

    public boolean isIfFeasible() {
        return ifFeasible;
    }

    public void setIfFeasible(boolean ifFeasible) {
        this.ifFeasible = ifFeasible;
    }

    public int getIndividualNo() {
        return individualNo;
    }

    public void setIndividualNo(int individualNo) {
        this.individualNo = individualNo;
    }


    public int getGenerationNo() {
        return generationNo;
    }

    public void setGenerationNo(int generationNo) {
        this.generationNo = generationNo;
    }

    public String print() {
        return "Record{" + "generationNo=" +
generationNo + ", individualNo=" + individualNo + ",
coverage=" + coverage + ", ifFeasible=" + ifFeasible +
'}';
    }
}

//Class RecordFile
package genut;

import genut.encoding.tree.NodeGene;
import genut.encoding.tree.TreeChromosome;
import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.InputStreamReader;
import java.util.*;

public class RecordFile
{
    Random ran;
    String fName;
    ArrayList<Record> records;
    RecordFile()
    {
        ran = new Random();
        fName = "C:\\mcs_file.txt";
        records = new ArrayList();
    }
    public TreeChromosome encodeTree()
    {
        TreeChromosome tree = new TreeChromosome();
        String tmp,tmp1,tmp2;
        String name;
        int x;
        int i;
        String type=null;
        ArrayList<String> tlist1 = new ArrayList();
        ArrayList<String> tlist2 = new ArrayList();
        try
        {
            FileInputStream fstream;
            fstream = new FileInputStream(fName);
            try (DataInputStream in = new
DataInputStream(fstream))
            {
                BufferedReader br = new
BufferedReader(new InputStreamReader(in));
                String strLine;
                while ((strLine = br.readLine()) != null)
                {
                    StringTokenizer st = new
StringTokenizer(strLine," (),");
                    x=0;
                    while (st.hasMoreTokens())
                    {
                        tmp = st.nextToken();

                        if("new".equals(tmp)) {
                            type = "constr";
                            x++;
                        }
                        if(x==1)
                        {
                            name=tmp;
                            x++;
                        }
```

```
                tlist1.add(tmp);
            }
            StringTokenizer sr = new
StringTokenizer(strLine,".()");
            while (sr.hasMoreTokens())
            {
                tmp = sr.nextToken();
                tlist2.add(tmp);
                type="method";
            }
            for(i=0;tlist1.get(i)!=null;i++)
            {
                tmp = tlist1.get(i);
                if("new".equals(tmp)) {
                    break;
                }
            }
            i++;
            NodeGene z1 = new
NodeGene(tlist1.get(i),type);
            tree.add(z1);
            while(tlist1.get(i)!=null)
            {
                i++;
                tmp1 = tlist1.get(i);
                i++;
                tmp2 = tlist1.get(i);
                i+=2;
                NodeGene z2 = new
NodeGene(tmp2,tmp1);
                tree.add(z2);
            }
          }
        }
      }catch (Exception e)
      {
         System.err.println("Error: " + e.getMessage());
      }
      return tree;
   }
   public int assgnintval()
   {
     int val = ran.nextInt();
     while((val<0)||(val>20)) //checking as per
specifications
     {
        val = ran.nextInt();
     }
     return val;
   }
   public float assgnfloatval()
   {
      float val = ran.nextFloat();
      while((val<0.0)||(val>20.0)) //checking as per
specifications
     {
        val = ran.nextInt();
     }
      return val;
   }
   public Boolean assgnboolval()
   {
      boolean val = ran.nextBoolean();
      return val;
   }

   void addRecord(Record record) {
      records.add(record);
   }

   void printRecords()
   {
      Record rec = records.remove(0);
      while(! records.isEmpty())
      {
        rec.print();
         rec = records.remove(0);
      }
   }
}

//Class ReplacementScheme
package genut;

import genut.encoding.tree.TreeChromosome;
import java.util.Arrays;
import java.util.Comparator;

public class ReplacementScheme
{
   @SuppressWarnings("unchecked")
   public TreeChromosome[]
replace(TreeChromosome[] parents,
TreeChromosome[] offsprings)
   {
      Comparator fitnessComp = new
FitnessComparator();
      Arrays.sort(parents, fitnessComp);
      Arrays.sort(offsprings, fitnessComp);

      TreeChromosome[] newGenerations = new
TreeChromosome [parents.length];
      int parentCounter = parents.length-1;
      int offspringCounter = offsprings.length-1;

      for (int newGenCounter=0; newGenCounter <
newGenerations.length; newGenCounter++)
      {
         if (offsprings[offspringCounter].fitness() <
parents[parentCounter].fitness())
         {
```

```
            newGenerations[newGenCounter] =
parents[parentCounter];
            parentCounter--;
         }
         else
         {
            newGenerations[newGenCounter] =
offsprings[offspringCounter];
            offspringCounter--;
         }
      }
      return newGenerations;
   }
}

//Class SelectionScheme
package genut;

import genut.encoding.tree.TreeChromosome;
import java.util.HashMap;
import java.util.Map;
import java.util.Random;

public class SelectionScheme
{
   public TreeChromosome[] select(TreeChromosome[]
population)
   {
      double sumFitness = 0.0;
      Map<TreeChromosome, Double> fitnessTable =
new HashMap();

      for(TreeChromosome c: population)
      {
         fitnessTable.put(c, c.fitness());
         sumFitness += fitnessTable.get(c);
      }

      Map<TreeChromosome, Double> sumEVTable =
new HashMap();
      sumEVTable.put( population[0], fitnessTable.get(
population[0] ) / sumFitness * population.length );
      for(int counter=1; counter < population.length;
counter++)
      {
         double ev = fitnessTable.get(
population[counter] ) / sumFitness * population.length;
         sumEVTable.put( population[counter],
sumEVTable.get( population[counter-1] ) + ev );

      }

      TreeChromosome[] selectedPopulation;
      selectedPopulation = new
TreeChromosome[population.length];
      Random randGenerator = new Random();

      for (int counter=0; counter <
selectedPopulation.length; counter++)
      {
         double randomValue =
randGenerator.nextDouble() * population.length;

         for (TreeChromosome c: sumEVTable.keySet())
         {
            if (randomValue  <= sumEVTable.get( c )  )
            {
               selectedPopulation[counter] = c;
            }
         }
      }
      return selectedPopulation;
   }
}

//Class Side
package genut;

class Side
{
   int length;
   Side() {
         length = 0;
   }
   Side(int x) {
      length = x;
   }
   void setSide(int x) {
      length = x;
   }
   int getSide() {
         return length;
   }
}

//Class Triangle
package genut;

class Triangle
{
   private Side side1;
   private Side side2;
   private Side side3;
   Triangle(int x,int y, int z)
   {
      side1 = new Side(x);
      side2 = new Side(y);
      side3 = new Side(z);
   }
   Side gets1()
   {
      return side1;
```

```
    }
    Side gets2()
    {
        return side2;
    }
    Side gets3()
    {
        return side3;
    }
    public void print()
    {
        System.out.println("Side1:"+side1.getSide()+"
Side2:"+side2.getSide()+" Side3:"+side3.getSide());
    }
}

//Class NodeGene


package genut.encoding.tree;
import java.util.ArrayList;

public class NodeGene implements Cloneable
{
    protected ArrayList<NodeGene> list;
    public String name;
    public String type;
    public String recr;
    public String recrType;
    public int intval;
    public float floatval;
    public Boolean boolval;
    public Object objval;
    private NodeGene parent;
    public NodeGene() {
        list = null;
    }
    public NodeGene(String na, String ty) {
        name = na;
        type = ty;
        list = new ArrayList();
    }
    public void addGene(NodeGene z) {
        list.add(z);
    }
    public void deleteGene(NodeGene z) {
        list.remove(z);
    }
    public NodeGene readGene(int index) {
        return (NodeGene)list.get(index);
    }
    public NodeGene getParent() {
            return parent;
    }
    public ArrayList<NodeGene> getGeneList()
    {
        return list;
    }
    public void deleteGeneList()
    {
        list.clear();
    }
    public boolean addGeneList(ArrayList<NodeGene>
newList)
    {
        return list.addAll(newList);
    }
    void setParent(NodeGene parent) {
        this.parent = parent;
    }
}

//Class NodeMutation

package genut.encoding.tree;

import java.util.Random;

public class NodeMutation
{
    public static void mutate(NodeGene gene)
    {
        Random randGenerator = new Random();
        if ( NodeGene.isVariable( gene ) )
        {
            if (randGenerator.nextBoolean())
            {
                gene.setValue(randGenerator.nextDouble() *
ConstrNodeGene.maxValue);
            }
            else
            {
                gene.setValue(randGenerator.nextDouble() *
ConstrNodeGene.maxValue * -0.1 );
            }
        }
        else
        {
            if (randGenerator.nextBoolean())
            {
                if (randGenerator.nextBoolean())
                {
                    gene.setValue(randGenerator.nextDouble()
* ConstrNodeGene.maxValue);
                }
                else
                {
                    gene.setValue(randGenerator.nextDouble()
* ConstrNodeGene.maxValue * -0.1 );
                }
            }
            else
```

```
            {

gene.setValue(ConstrNodeGene.VARIABLE);
            }
        }
    }
    public static void mutate(MethodNodeGene gene)
    {
        Random randGenerator = new Random();
        int mutatedValue =
randGenerator.nextInt(MethodNodeGene.MAX_NUM
BER_OF_OPERATOR);

        while (mutatedValue == gene.getValue())
        {
            mutatedValue =
randGenerator.nextInt(MethodNodeGene.MAX_NUM
BER_OF_OPERATOR);
        }

        gene.setValue(mutatedValue);
    }
}

//Class TreeChromosome

package genut.encoding.tree;

import java.util.ArrayList;
import java.util.Random;

public class TreeChromosome
{

    private static NodeGene
createRandomNode(NodeGene parent) {
        throw new UnsupportedOperationException("Not
yet implemented");
    }
    private ArrayList<NodeGene> list;
    double coverage;
    boolean ifFeasible;
    private NodeGene root;
    private TreeFitnessFunction fitnessFunction;
    private TreeCrossover treeCrossover;
    private double fitness;
    private static Random randGenerator = new
Random();
    private static int maxOperatorNodes = 15;
    private static int currentMaxOperatorNodes;
    private static int usedOperatorNodes;
    public double getCoverage()
    {
        return coverage;
    }
    public TreeChromosome()
    {
        list = new ArrayList();
        fitness = (float)0.0;
        coverage = (float)0.0;
    }
    public TreeChromosome(NodeGene root,
TreeFitnessFunction fitnessFunction)
    {
            setTreeFitnessFunction(fitnessFunction);
            setRoot(root);
    }

    public void setFeasible(boolean b)
    {
        ifFeasible = b;
    }
    public void setFitness(double fit)
    {
        fitness = fit;
    }
     public void setCoverage(double cov)
    {
        coverage = cov;
    }
    public void add(NodeGene z)
    {
        list.add(z);
    }
    public void delete(NodeGene z)
    {
        list.remove(z);
    }
    public ArrayList getList()
    {
        return list;
    }
    public NodeGene getElement(int index)
    {
        return (NodeGene)list.get(index);
    }

    public double fitness() {
        return fitness;
    }

    public void setRoot(NodeGene root)
            {
                    if (root != null)
                    {
                            this.root = root;
                    }

                    calcFitness();
            }
```

```
	public void
setTreeFitnessFunction(TreeFitnessFunction
fitnessFunction)
	{
		if (fitnessFunction != null)
		{
			this.fitnessFunction =
fitnessFunction;
			calcFitness();
		}
	}

	public void setTreeCrossover(TreeCrossover
treeCrossover)
	{
		if (treeCrossover != null)
		{
			this.treeCrossover =
treeCrossover;
		}
	}

	public NodeGene getRoot()
	{
		return root;
	}

	public TreeFitnessFunction
getTreeFitnessFunction()
	{
		return fitnessFunction;
	}

	public TreeCrossover getTreeCrossover()
	{
		return treeCrossover;
	}

	public TreeChromosome[]
crossover(TreeChromosome pair)
	{
		if (treeCrossover == null)
		{
			treeCrossover = new
TreeCrossover(fitnessFunction);
		}
	 return treeCrossover.crossover(this, pair);
	}

	private static void
createRandomChildren(NodeGene parent)
	{
		if (usedOperatorNodes <
currentMaxOperatorNodes)
		{
			NodeGene leftChild =
createRandomNode(parent);

			if (leftChild instanceof
NodeGene)
			{

	usedOperatorNodes++;

	createRandomChildren( (NodeGene) leftChild
);
			}

			if (usedOperatorNodes <
currentMaxOperatorNodes)
			{
				NodeGene
rightChild = createRandomNode(parent);

				if (rightChild
instanceof NodeGene)
				{

	usedOperatorNodes++;

	createRandomChildren( (NodeGene)
rightChild );
				}
			}
		}
	}

   private void calcFitness() {
      throw new UnsupportedOperationException("Not
yet implemented");
   }

   public boolean getFeasible() {
      return ifFeasible;
   }

   public void mutate(double mutation_rate) {
      throw new UnsupportedOperationException("Not
yet implemented");
   }
}

//Class TreeCrossover
package genut.encoding.tree;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class TreeCrossover
{
```

```
        private Random randGenerator;
        private TreeFitnessFunction fitnessFunction;

        public TreeCrossover(TreeFitnessFunction
fitnessFunction) {
                this.fitnessFunction = fitnessFunction;
        }

        public TreeChromosome[]
crossover(TreeChromosome parentA, TreeChromosome
parentB)
        {
      TreeChromosome[] offsprings = new
TreeChromosome[2];
      NodeGene rootDuplicateA = null;
      NodeGene rootDuplicateB = null;

      List<NodeGene> duplicateListA = new
ArrayList();
      List<NodeGene> duplicateListB = new
ArrayList();
      duplicateListB.add(rootDuplicateB);

      randGenerator = new Random();
      NodeGene crossPointA = duplicateListA.get(
randGenerator.nextInt(duplicateListA.size()-2) + 1 );
      NodeGene crossPointB = duplicateListB.get(
randGenerator.nextInt(duplicateListB.size()-2) + 1 );
      NodeGene tempParent = (NodeGene)
crossPointB.getParent();

      if(crossPointA.type == null ? crossPointB.type
== null : crossPointA.type.equals(crossPointB.type))
      {
        ArrayList<NodeGene> tempList1 =
crossPointA.getParent().getGeneList();
        crossPointA.getParent().deleteGeneList();
        ArrayList<NodeGene> tempList2 =
crossPointB.getParent().getGeneList();
        crossPointB.getParent().deleteGeneList();


crossPointA.getParent().addGeneList(tempList2);

crossPointB.getParent().addGeneList(tempList1);

        crossPointA.setParent(tempParent);

        offsprings[0x0] = new
TreeChromosome((NodeGene) rootDuplicateA,
fitnessFunction);
        offsprings[0x1] = new
TreeChromosome((NodeGene) rootDuplicateB,
fitnessFunction);
        return offsprings;
      }
      return crossover(parentA, parentB);
       }
}
```

List of Publications by Author

Journals

1. Gupta N., Saini D. and Saini H.,"Class Level Test Case Generation in Object Oriented Software Testing", International Journal of Information Technology and Web Engineering, IGI Publishing, Hershey, Pennsylvania, USA, Vol. 3, No. 2, pp.19-26, April-June 2008.
2. Gupta N. and Rohil M., "Using Genetic Algorithm for Unit Testing of Object Oriented Software", International Journal of Simulation: Systems, Science & Technology, EDAS London, Vol. 10, No. 3, pp. 97-102,May 2009.
3. Gupta N. and Rohil M., "An Approach for Detection and Correction of Design Defects in Object Oriented Software" International Journal of Information Technology & Knowledge Management (ISSN: 0973-4414), Volume-4, Number-I, pp.63-67, January-June 2011.
4. Gupta N. and RohilM., "Measuring Maintenance Effort in Object Oriented Software with Indirect Coupling", International journal of Computer Applications, Foundation of Computer Science, USA, Vol. 54, No. 2, pp.19-24, September 2012.

Conferences

1. Gupta N. and Rohil M. "Using Genetic Algorithm for Unit Testing of Object Oriented Software", Proceedings of the International Conference on Emerging Trends in Engineering and Technology, 2008. ICETET '08 on 16-18 July 2008, Nagpur (India), IEEE Computer Society, pp. 308 – 313, July 2008.
2. Gupta N. and Rohil M. "Issues and Problems in Generation of Automated Test Data for Object Oriented Systems", Proceedings of IEEE International Advance Computing Conference (IACC 2009) Patiala, India, 6–7 March 2009, IEEE Computer Society, pp.1868 – 1871, March 2009.
3. Gupta N. and Rohil M. "Exploring Possibilities of Reducing Maintenance Effort in Object Oriented Software by Minimizing Indirect Coupling", Proceedings of the 2nd International Conference on Computer Science, Engineering & Applications (ICCSEA 2012), May 25-27, 2012, New Delhi, India, Published by Springer in Advances in Intelligent and Soft Computing, Vol. 166, pp.959-965, May 2012.
4. Gupta N. and Rohil M., "Object Oriented Software Maintenance in presence of Indirect Coupling", Proceedings of the 5th International Conference on Contemporary Computing, August 6-8, 2012, Noida, India, Published by Springer in Communications in Computer and Information Science Vol. 306, pp. 442 – 451, August 2012.
5. Gupta N. and Rohil M., "Software Quality Measurement for Reusability", In Proceedings of International Conference on Software Engineering and Mobile Application Modelling and Development, 19-21 Dec 2012, Chennai, pp. 49-61, December 2012.
6. Gupta N. and Rohil M., "Improving GA based Automated Test Data Generation Technique for Object Oriented Software", In Proceedings of 3rd IEEE International Advance Computing Conference (IACC-2013), 22-23 Feb. 2013, Ghaziabad, Print ISBN: 978-1-4673-4527-9, pp. 249 – 253, 2013.

Brief Biography of the Candidate

Nirmal Kumar Gupta received his B.E. in Electronics and Instrumentation Engineering from Bundelkhand Institute of Engineering and Technology, Jhansi – 284001 (Uttar Pradesh), Indiain 1998 and M.Tech degree in Computer Science & Technology from Indian Institute of Technology Roorkee, Roorkee – 247667 (Uttarakhand) India in 2002.

He served at Babu Banarasidas National Institute of Technology and Management, Lucknow as Lecturer during Apr 2002 to Jan 2004, at Institute of Technology and Management, Gurgaon during Feb 2004 to Dec 2004, as Sr. Lecturer and since Jan 2005 he is working as Lecturer with Department of Computer Science and Information Systems, Birla Institute of Technology and Science, Pilani – 333031, Rajasthan, India. His area of research interest includes Software Engineering and Testing, Object Oriented Technologies and Soft Computing Techniques. He has published 10 research papers in conferences and in Journals of international repute.

Brief Biography of the Supervisor

Mukesh Kumar Rohil received the M.Sc. (Hons) degree in Physics, the B.E. (Hons) degree in Civil Engineering, the M.E. degree in Systems & Information and the Ph.D. degree in Computer Science & Engineering from the Birla Institute of Technology and Science, Pilani – 333 031, Rajasthan, India, in 1993, 1993, 1995 and 2004 respectively.

He served M/s Asia Polytex India Limited, Mumbai (earlier Bombay) as Systems Analyst-cum-Programmer during Aug-Nov, 1993. He worked as ASSISTANT LECTURER during June 1995 and Dec 1998, as Lecturer during Jan 1998 and May2006, Assistant Professor from June 2006 to January 2013 and since February 2013 he is working as Associate Professor with Computer Science and Information Systems Group, Birla Institute of Technology and Science, Pilani – 333 031, Rajasthan, India. His area of research interest includes artificial intelligence and expert systems, computer graphics, digital image processing of remote sensing data, GIS, Applications of Artificial Intelligence in Software Engineering and pattern recognition.

Dr. Rohil has published 29 articles in conferences, journals and book-chapters. Dr. Rohil is a life member of Indian Society of Remote Sensing (ISRS). Dr. Rohil received OPTO-MECH award for the best paper presented titled "Exploring possible applications of fuzzy logic in simulation of three-dimensional visualization using remote sensing data" presented in annual convention of ISRS, in Nov 2000. Dr. Rohil has received commendation certificate for the article "Digital Imaging in Teaching Construction Process" presented in 6th Annual Convention & Seminar on "Education and Training of Building Professionals", New Delhi; April 20-22, 2000 & published in the Journal of Indian Buildings Congress; New Delhi; April 2000, 7:1, 47-53