An Efficient Approach to Identify Key Classes of Software to Assist Initial Program Comprehension

By

Muhammad Kamran

(2009-NUST-MS PhD-CSE (E)-04)

Submitted to the Department of Computer Engineering
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Software Engineering

Advisor
Dr. Farooque Azam

College of Electrical & Mechanical Engineering
National University of Sciences and Technology
2013

APPROVAL

It is certified that the contents and form of thesis entitled **"An Efficient Approach to Identify Key Classes of Software to Assist Initial Program Comprehension"** submitted by **Muhammad Kamran**, have been found satisfactory for the requirement of degree.

Advisor: ___________________
(Dr. Farooque Azam)

Committee Member: __________________
(Dr. Aasia Khanum)

Committee Member: __________________
(Dr. Arslan Shaukat)

Committee Member: __________________
(Dr. Muhammad Abbas)

IN THE NAME OF ALMIGHTY ALLAH
THE MOST BENEFICENT AND THE MOST MERCIFUL

TO MY PARENTS, WIFE AND SON

Certificate of Originality

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any degree or diploma at NUST CEME or at any other educational institute, except where due acknowledgement has been made in the thesis. Any contribution made to the research by others, with whom I have worked at NUST CEME or elsewhere, is explicitly acknowledged in the thesis.

I also declare that the intellectual content of this thesis is the product of my own work, except for the assistance from others in the project's design and conception or in style, presentation and linguistics which has been acknowledged.

Author Name: Muhammad Kamran

Signature: ______________

ACKNOWLEDGEMENTS

First of all I am extremely thankful to Almighty Allah for giving me courage and strength to complete this challenging task and to compete with international research community. I am also grateful to my family, especially my parents who have supported and encouraged me through their prayers that have always been with me.

I am highly thankful to Dr. Farooque Azam for his valuable suggestions and continuous guidance throughout my research work.

I am highly grateful to the committee members for their help and guidance throughout the research work. I am also thankful to all of my teachers who have been guiding me throughout my course work and have contributed to my knowledge. Their knowledge, guidance and training helped me a lot to carry out this research work.

I am also thankful to Mr. Amer Shahzad for his keen interest, guidance and feedback in this research work. I would like to offer my gratitude to all my close colleagues who have been encouraging me throughout my research work especially Mr. Muhammad Shahzad.

Muhammad Kamran

ABSTRACT

The constant modification of software systems, the increasing size of the software and the expensive development process are the factors that are responsible for the increase in the amount of the effort that is being expended on the maintenance phase. Mostly each maintenance cycle is performed to achieve a specific goal, for instance improving the efficiency of a procedure, provision of new application features, assembling existing components into the new software and so on. A programmer will not be able to achieve any of the above goals unless he understands the particular software at a sufficient level of detail that allows him to implement the desired change in the system. The process of exploring the software and its associated artifacts with the aim to gain knowledge about the inner workings of the system for carrying out the necessary changes in the system is termed as program comprehension.

The process of building an understanding of the existing system is time consuming and takes around 40% of the allocated time for a maintenance task. How a new software developer proceeds to build an initial acquaintance with the original software, differs a lot and depends on a number of factors like the experience of the individual, the size and type of the software, the level of detail required to modify the system and so on. Numerous efforts have been made to reduce the time consumed in the program comprehension process by providing support to the programmer during this phase. The key contribution of this thesis is a heuristic approach that can aid the programmer in searching the key classes of software that are ideal nominees for the initial stages of the program comprehension process.

Full automation of the process of developing an understanding of the program is not possible since it involves human learning activity. Therefore, it has been suggested that the specialized tools should help the programmer discover the software amicably. The segments of the program that can be attractive from the comprehension viewpoint must be brought into the notice of the programmer by the exploration tools. In our case the program discovery tool should pinpoint the key classes of the object-oriented system that are fundamental to its design.

TABLE OF CONTENTS

ABSTRACT V

TABLE OF CONTENTS VI

LIST OF PUBLICATIONS X

LIST OF FIGURES XI

LIST OF TABLES XII

CHAPTER 1 1

INTRODUCTION 1

1.1 MOTIVATION 1

1.2 BACKGROUND 2

1.3 PROBLEM STATEMENT 2

1.4 PROPOSED SOLUTION 3

1.4.1 DYNAMIC COUPLING BASED SOLUTION 3

1.5 PUBLICATION 4

1.6 ORGANIZATION OF THE THESIS 4

CHAPTER 2 5

LITERATURE REVIEW 5

2.1 PROGRAM COMPREHENSION 5

2.2 MODELS OF PROGRAM COMPREHENSION 7

2.2.1 TOP-DOWN MODEL OF PROGRAM COMPREHENSION 7

2.2.2 BOTTOM-UP MODEL OF PROGRAM COMPREHENSION 8

2.2.3 INTEGRATED MODEL 9

2.3 PROGRAM ANALYSIS APPROACHES FOR COMPREHENSION 10

2.3.1 DYNAMIC ANALYSIS VS. STATIC ANALYSIS 10

2.4 THE DYNAMIC ANALYSIS APPROACH FOR PROGRAM COMPREHENSION 10

2.4.1 DYNAMIC ANALYSIS FACILITATES TARGET-ORIENTED COMPREHENSION 10

2.4.2 DYNAMIC ANALYSIS REVEALS ACTUAL PICTURE OF POLYMORPHISM (LATE BINDING) 11

2.4.3 Technological Support for Dynamic Analysis of Program 12
2.4.4 The Consequences of Observing – Observer Effect 13
2.4.5 The perils of Dynamic Analysis 14

2.5 USE OF COUPLING FOR PROGRAM COMPREHENSION 15

2.5.1 What is Coupling? 15
2.5.2 Why dynamic coupling metrics? 16
2.5.3 Possible variations for calculating dynamic coupling metrics 16
2.5.4 How the dynamic coupling metrics are calculated 19

2.6 KEY CLASSES OF SOFTWARE 22

2.6.1 Existing Approaches to Identify Key Classes of Software 23

CHAPTER 3 27

OVERVIEW OF TECHNOLOGIES USED IN THIS RESEARCH 27

3.1 INTRODUCTION TO THE TECHNOLOGY USED FOR DYNAMIC ANALYSIS 27

3.1.1 An overview of AspectJ 27
3.1.2 What are Joinpoints? 28
3.1.3 The role of Pointcuts 28
3.1.4 Advices 33
3.1.5 The role of Aspects 34
3.1.6 What is Load-Time Weaving (LTW)? 34

CHAPTER 4 37

RESEARCH METHODOLOGY 37

4.1 RESEARCH METHODS 37

4.1.1 Conducting a Survey 37
4.1.2 Controlled Experimentation 38
4.1.3 Case Study 38

4.2 THE ORGANIZATION OF THE EXPERIMENTAL SYSTEM 39

4.2.1 Selection criteria for the case study 39
4.2.2 Selection of Use Case for Tracing 39
4.2.3 The baseline for the experiment 40
4.2.4 Outcome 40

4.3 OVERVIEW OF THE 1ST CASE STUDY -- APACHE ANT 41
4.3.1 WHAT IS APACHE ANT? 41
4.3.2 USE CASE FOR TRACING 41
4.3.3 ARCHITECTURE OF ANT 42
4.4 OVERVIEW OF THE 2ND CASE STUDY – JAKARTA JMETER 44
4.4.1 WHAT IS JAKARTA JMETER? 44
4.4.2 USE CASE FOR TRACING 45
4.4.3 ARCHITECTURE OF JMETER 45
CHAPTER 5 47
THE PROPOSED APPROACH 47
5.1 PROBLEMS WITH EXISTING APPROACHES TO IDENTIFY KEY CLASSES 47
5.2 BASIC IDEA BEHIND OUR APPROACH 48
5.2.1 HYPOTHESIS 48
5.3 IMPLEMENTATION 53
5.3.1 START THE SYSTEM USING ASPECTJ LIBRARY 53
5.3.2 RUN THE INSTRUMENTED SYSTEM AND CALCULATE METRICS 54
5.3.2.1 Algorithm for Calculation of Dynamic Coupling Metric 55
5.3.2.2 Flow of activities for calculation of Dynamic Coupling Metric 57
5.3.3 RANK THE RESULTS 58
5.3.4 SYSTEM REQUIREMENTS 58
5.4 HOW THE PROPOSED APPROACH IS EVALUATED AND VALIDATED 58
5.5 PRACTICAL USE OF OUR APPROACH 59
CHAPTER 6 60
RESULTS AND EVALUATION 60
6.1 RESULTS OF 1ST CASE STUDY – APACHE ANT 60
6.2 EVALUATION OF 1ST CASE STUDY – APACHE ANT 62
6.2.1 CRITERIA USED FOR EVALUATION 62
6.2.2 PRECISION COMPARISON 63
6.2.3 RECALL COMPARISON 64
6.2.4 FALLOUT COMPARISON 64

6.2.5 F-measure Comparison 65
6.2.6 Time Usage – (effort to perform the complete analysis, from start to finish) 66
6.3 RESULTS OF 2ND CASE STUDY – JAKARTA JMETER 68
6.4 EVALUATION OF 2ND CASE STUDY – JAKARTA JMETER 69
6.4.1 Precision Comparison 69
6.4.2 Recall Comparison 70
6.4.3 Fallout Comparison 70
6.4.4 F-measure Comparison 71
6.4.5 Time Usage – (effort to perform the complete analysis, from start to finish) 72
6.5 AN OUTLINE OF ACHIEVEMENTS IN THIS RESEARCH 73
CHAPTER 7 74
CONCLUSION AND FUTURE WORK 74
7.1 CONCLUSION 74
7.2 CONTRIBUTION 75
7.3 FUTURE WORK 76

LIST OF PUBLICATIONS

[1] Muhammad Kamran, Farooque Azam, Asia Khanum, **"Discovering Core Architecture Classes to Assist Initial Program Comprehension"**, Lecture Notes in Electrical Engineering Volume 211, 2013, pp 3-10, Springer Berlin Heidelberg.

LIST OF FIGURES

Figure 1: A Simple Sequence Diagram 19
Figure 2: Another Simple Sequence Diagram 20
Figure 3: Slightly Complicated Sequence Diagram 21
Figure 4: Key Classes of Object-Oriented Software [8] 23
Figure 5: Using WMC and DAC to identify Key Classes [23] 24
Figure 6: A Compacted Call Graph [24] 24
Figure 7: HITS Webmining Algorithm [25] 25
Figure 8: Indirect Coupling example [12] 25
Figure 9: Simple Advice example before/after call of public method 33
Figure 10: Advice example that uses context collected by target 34
Figure 11: Class diagram showing 5 important classes of Apache Ant 43
Figure 12: Core Structure Analogy from Civil Engineering 48
Figure 13: Sample Sequence Diagram to calculate IC_CMS 50
Figure 14: Slightly Complicated Sequence Diagram to Calculate IC_CMS 51
Figure 15: Overview of the Approach 53
Figure 16: Load-time weaving of Application Classes using aspectjweaver.jar 54
Figure 17: Flow of activities for calculation of dynamic coupling metric 57
Figure 18: Comparison of Precision % -- Apache Ant 63
Figure 19: Comparison of Recall % -- Apache Ant 64
Figure 20: Comparison of Fallout % -- Apache Ant 65
Figure 21: Comparison of F-Measure % -- Apache Ant 66
Figure 22: Comparison of time usage of approaches – Apache Ant 67
Figure 23: Comparison of Precision % -- Jakarta JMeter 69
Figure 24: Comparison of Recall % -- Jakarta JMeter 70
Figure 25: Comparison of Fallout % -- Jakarta JMeter 71
Figure 26: Comparison of F-Measure % -- Jakarta JMeter 72
Figure 27: Comparison of time usage of approaches – Jakarta JMeter 73

LIST OF TABLES

Table 1: Tasks and activities requiring code understanding [6] 6
Table 2: Variations of Dynamic coupling measures [10] 17
Table 3: Dynamic Coupling Metrics [10] 18
Table 4: Calculating Dynamic Coupling Metrics – Simple Sequence Diagram...... 19
Table 5: Calculating Dynamic Coupling Metrics – Only Two Classes Involved.... 20
Table 6: Calculating Dynamic Coupling Metrics – Complex Sequence Diagram .. 22
Table 7: Summary of existing approaches to identify Key Classes of Software 26
Table 8: Pointcuts for call to constructors/methods 28
Table 9: Poincuts for execution of constructors/methods 29
Table 10: Dynamic coupling measures [10] 49
Table 11: Calculating IC_CM and our extended version of IC_CM 50
Table 12: Calculating Dynamic Coupling Metrics 52
Table 13: System Specifications 58
Table 14: Classes identified by our approach (Top 15%) – Apache Ant................. 61
Table 15: Comparison of Precision % -- Apache Ant.. 63
Table 16: Comparison of Recall % -- Apache Ant 64
Table 17: Comparison of Fallout % -- Apache Ant 64
Table 18: Comparison of F-Measure % -- Apache Ant 65
Table 19: Details of results for Time Usage – Apache Ant 67
Table 20: Classes identified by our approach (Top 15%) – Jakarta JMeter 68
Table 21: Comparison of Precision % -- Jakarta JMeter 69
Table 22: Comparison of Recall % -- Jakarta JMeter 70
Table 23: Comparison of Fallout % -- Jakarta JMeter 70
Table 24: Comparison of F-Measure % -- Jakarta JMeter 71
Table 25: Details of results for Time Usage – Jakarta JMeter 73

CHAPTER 1

INTRODUCTION

An introduction to the research work that has been taken in this thesis is presented in this chapter. It includes motivation and definition of the problem. Moreover the objectives and goals are also discussed.

1.1 Motivation

Software development of an entirely new project is amusing. You can utilize the power of your ingenuity when building the software from scratch. You are free to decide the variable set of parameters that include the hardware and software architecture of the system, the choice of the technology set, the design patterns to be followed and so on. Moreover, the choice of the programming language can be made based on the available expertise and the ease of development offered by the language platform.

Since change is the only constant in this world, therefore changes are inevitable in the environment in which the software operates. Software must be changed in response to the changes in the environment to fulfill the expectations of the users and to avoid the threat of being outdated. The maintenance phase of already built software brings more miseries to the life of the software developers as compared to the development of a new software project.

The most challenging and the most expensive (in terms of time usage) part of software maintenance is the effort to build an understanding of the existing system, also known as program comprehension. Usually the documentation and other associated software artifacts are utilized for this process, but in many cases these artifacts are either not available or they do not reflect the current state of the system. To help the software developer in this kind of a situation, a solution is discussed in this thesis that can be used in the early stages of program comprehension process. Our

solution is based on the dynamic analysis of the software, in other words we collect the information from the running software system.

1.2 Background

The programmer needs to build a perceptive association in his mind when programming a piece of software. This association binds the code that has been written by him and the system actions he wants to program [19]. On the other hand, while trying to get acquaintance with the system, in fact a programmer is struggling to figure out the reverse mapping: i.e. building an association between the external behavior (or functionality) of the system and the underlying code that is responsible for that behavior.

The literature suggests that the program comprehension phase is known to consume between 30 to 60% (depending on the source) of a software engineer's time. To build an adequate understanding of a software system, the programmer needs to study the program code, associated software artifacts and related documents in the program comprehension phase [2, 3]. The adequate level of understanding is identified as a level where the programmer knows that the system architecture, design or functionality will not be hurt by the change that he is making.

1.3 Problem Statement

It has been demonstrated by empirical studies that the majority of experienced developers track the structural dependencies in the source code when they need to derive the high-level model related with the upfront task [7].

On the other side, many greenhorn developers, employed to an unknown system might be trapped in insignificant code easily and do not succeed in finding the vital program functionality, that could result in low trait software maintenance [7] or wastage of time.

In this research, we aim to reduce the problem of program comprehension by providing the programmer with a small number of preliminary classes, which can be utilized to begin the tracking of structural dependencies in order to gain familiarity

with the system. Our main goal is to devise an efficient approach that can assist the software developer in the initial stages of program comprehension process.

1.4 Proposed Solution

We propose that the key classes of the software are the ideal candidates to start the program comprehension efforts as these classes implement the key concepts of the system. We have devised an efficient approach that identifies the key classes of software using an extended version of dynamic coupling metric.

1.4.1 Dynamic coupling based solution

The alleviation of the afore-mentioned problems has been taken in this thesis. The underlying thought for utilizing the coupling is the fact that the modules that are considered important for first round of program comprehension process can be pointed out by structural dependencies that are present in the system [15]. We have used a novel variant of a dynamic coupling metric, which requires an execution scenario offering good code coverage. The dynamic coupling metric provides us with all interactions that take place at runtime. A high count of coupling in a module indicates that it requests other modules to do majority of work (delegation) and often serves as a part of the core structure.

Typically the coupling is measured between two classes/modules of the system, whereas we are interested in the discovering the entire structural topology (specifically the core part of the structure) of the application. We have used AspectJ for run-time analysis of Java applications to retrieve key classes of system by calculating a novel variant of dynamic coupling metric. The key classes could reveal important structural properties of the system and hence are strong candidates for the early program comprehension process. They can be a good starting point to understand additional classes and their connections.

The outcome of our heuristic is a ranked list containing all those classes of the system that were instantiated or whose method(s) were called during the execution scenario. The ranking of the classes is done on the basis of our variant of dynamic

coupling metric that takes into account the loading order of the class in the application. In simple words, the classes are ranked based on their relevance to the initial stages of program comprehension. The validation of our approach is done using two open source systems as case study, namely Apache Ant 1.6.1 and Jakarta JMeter 2.0.1. The results are compared with the results of other analogous experiments performed on the same Guinea Pig Systems (Apache Ant 1.6.1 and Jakarta JMeter 2.0.1).

1.5 Publication

This thesis has produced the following publication:

- Muhammad Kamran, Farooque Azam, Assia Khanum, *"Discovering Core Architecture Classes to Assist Initial Program Comprehension"*, Lecture Notes in Electrical Engineering Volume 211, 2013, pp 3-10, Springer Berlin Heidelberg

1.6 Organization of the thesis

The rest of the document is organized as follows:

Chapter 2 is allocated to literature review which discusses different concepts related to Program Comprehension. The existing approaches for discovering key classes are also described. In Chapter 3 overview of the technologies used in this thesis is provided, Chapter 4 discusses the research methodology followed by us. In Chapter 5, the proposed approach is presented and its implementation details are provided. Chapter 6 discusses evaluation and results of the implemented system, and in Chapter 7 thesis work is concluded and some future directions for research are discussed.

CHAPTER 2

LITERATURE REVIEW

2.1 Program Comprehension

When a programmer is at the start of building an initial knowledge base of a new system or a subpart of the system, he has to construct an informal, human oriented view of the objectives of the system. The formation of this view happens in the course of scrutiny, experimentation, deduction and jigsaw-like assembly [4].

As far as the definition of program comprehension is concerened, we stick to the definition introduced by [4]:

"A person understands a program when able to explain the program, its structure, its behavior, its effects on its operation context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program."

From the above definition, it can be figured out that the program comprehension process is correlated to the problem of concept assignment. For a given program, the individual programmer discovers the human oriented concepts and maps them to their implementation oriented equivalents [7]. It becomes obvious that the program comprehension is an extremely individual process. The results differ greatly from one programmer to another, even when the understanding of the software is being built in the same way.

The process of building an understanding of a software system is different for individual programmers. Past experience of the similar nature, the level of knowledge required for the upfront task, the sheer size of the system to be studied, the programming language used for the system, prior knowledge of the system under study, etc. are the factors that affect the program understanding process [5]. Theory suggests that it is essential to comprehend the whole system prior performing any modifications, but practical experiences dictate that it is preferable to utilize a target oriented approach or need-based ploy. That means you are specifically interested in

obtaining the understanding of the particular subset of the program that is relevant to the maintenance task you have. Furthermore, economical constraints imply that this knowledge should be gained rapidly and comprehensively.

The first thing that we will try to explain in this chapter is the necessity of program understanding in the maintenance/reengineering phase, after that some prominent program comprehension theories will be discussed.

The need for program understanding: The understanding of the program is a necessary requirement for the software (re)engineering activities. Authors in [6] have made a list of specific scenarios in software maintenance for which program comprehension is a compulsory requirement [6]. An overview of these maintenance activities is provided in Table 1.

It is evident from Table 1 that majority of software maintenance activities in routine necessitate an adequate level of knowledge of the application to be maintained. The link between software evolution and program comprehension is established by the fact that majority of software evolution activities involve prior understanding of the software system.

Since we know that most software maintenance/reengineering processes include a prior program comprehension phase that can consume up to sixty percent of the programmer's time [1, 3], hence the improvement in the efficiency of this phase can increase the overall efficiency significantly.

Table 1: Tasks and activities requiring code understanding [6]

Maintenance Tasks	Activities
Adaptive	1) Gain System Understanding 2) Define requirements for adaptation 3) Develop adaptation design (preliminary and detailed) 4) Change Code, Debugging, Regression testing
Perfective	1) Gain System Understanding 2) Diagnose/Define requirements for improvements 3) Develop perfective design (preliminary and detailed) 4) Change Code, Debugging, Regression testing
Corrective	1) Gain System Understanding 2) Produce/Assess problem hypotheses 3) Fix the code, Regression testing

Reuse	1) Gain Problem Understanding 2) Search for a ‘close fit’ solution based on reusable components 3) Obtain reusable components and perform Integration
Code leverage	1) Gain Problem Understanding 2) Search for a ‘predefined components’ based solution 3) Organize solution to raise chances of predefined component usage 4) Obtain predefined components and perform Integration

2.2 Models of Program Comprehension

It has already been mentioned that program comprehension is an extremely individual process. How a software engineer performs and achieves the targets of his program understanding process is influenced by a number of factors. Some of the factors (which can very subjective at times) have been mentioned by [6] are listed below:

- The experience of the programmer
- Familiarity with the similar solution
- The complexity of the system
- The level of familiarity with the problem domain
- The amount of time available

The existing approaches for the program comprehension process have been extracted from the studies that lie on the border line of psychology and computer science. Primarily, these approaches can be broken down into three models of program comprehension: i) the top-down model, ii) the bottom-up model, iii) the integrated model (a hybrid of the preceding two models) [6]. Now we will explain each of these models.

2.2.1 Top-down model of program comprehension

When the code of the software system, its problem domain and/or associated solution space is known to the programmer, top-down understanding is typically applied [6]. This originates from the thought that if a piece of code, that performed the identical or comparable tasks, has already been learnt by the software engineer, then the structure of the code will have similarities. In a top-down model of program

understanding, these similarities in the structure of the code are straightforward to identify.

In the top-down model, typically a software engineer already has a perceived sketch of the structure of the software in his mind when he goes about his program understanding process. This perceived sketch can come from prior encounters with similar system in the same domain, utilizing comparable technology set, etc. or from suggested best practices/guidelines in the coding, documentation or associated artifacts. The ability to associate the current solution with previously seen construct based on a trigger of memory is called a beacon in program comprehension terminology. An excellent example of a beacon in software engineering terminology is design pattern, e.g. a DAO (Data Access Object) pattern, which is a signal about the organization of the persistence layer in the system.

In the top-down program comprehension model, the perceived sketch of the structure of the system is concluded in multiple rounds using hypotheses and the results of hypotheses about the software system. The refinement of hypotheses is done in iterations, after going through a number of levels, until a match is found in the associated artifacts (specific entity in the program code, documentation, or configuration file) [6].

2.2.2 Bottom-up model of program comprehension

The non-familiarity with the code and/or problem domain frequently results in the choice of bottom-up comprehension model by the software engineer. The two variations of the bottom-up model are described in this section.

Program model: The authors in [20] found that the initial picture built in mind when code is completely new to programmers, is the control-flow abstraction of program known as program model. This initial picture, assembled by means of beacons in the bottom-up fashion, discovers the major portions of code in the program. The program model is formed by the amalgamation of microstructures into macrostructures and by means of cross-referencing. Creation of bigger entities from small blocks for explanation is amalgamation, whereas cross-referencing builds a

relationship of these bigger entities with a higher abstraction level. The example of cross-referencing can be the chunking of all classes working jointly to form a linked list. Then understanding the intention behind this linked list is cross-referencing this list to more abstract level.

Situation model: The situation model is another model that authors in [20] have identified. This model provides an abstraction of dataflow/functionality while operating in a bottom-up manner. The understanding of the currently implemented practices/domain of the system is a pre-requisite for the applicability of this model. A simple instance of such kind of program comprehension in bottom-up manner is relating the statement in code *stockCount = stockCount - salesCount* to a descriptive statement like "reducing the stock count by the number of sales count". The amalgamation of situation knowledge at lower level with the situation knowledge at higher level can be exercised here as well. As soon as the goal of program is achieved, the situation model is ended.

2.2.3 Integrated model

The top-down model, bottom-up models (program/situation model) and a knowledge base constitute the integrated model for code comprehension. Usually, the human memory serves as the knowledge base and it serves as storage for the following:

(1) All the knowledge that is gained directly by applying the program comprehension strategies (top-down or bottom-up)

(2) Any derived or indirectly obtained information.

The most frequently applied model in practice when trying to comprehend industrial systems is the integrated model. This is because of the fact that particular segments of the code may be recognizable by the programmer due to past encounters while there may be segments in the code that are entirely novel.

2.3 Program Analysis Approaches for Comprehension

2.3.1 Dynamic Analysis vs. Static Analysis

The examination of the properties of the system (that needs to be studied) using the knowledge acquired from the running system is known as *dynamic analysis*, in software engineering.

The *static analysis* is positioned opposite to the concept of dynamic analysis. In static analysis the knowledge about the system is gathered using the artifacts such as the source code, documentation related to the system, configuration files, etc. to examine the properties of the system.

The execution trace of the software system needs to be generated as a pre-requisite to enable dynamic analysis of the software. The structure for the storage of the runtime information is referred as the execution trace. The software system needs to be executed using a well-defined execution scenario to record the execution trace. An execution scenario can be defined as an instance of interaction with the system covering a subset of use case(s).

2.4 The Dynamic Analysis Approach for Program Comprehension

The dynamic analysis approach is preferred over the static analysis approach for program comprehension because of the two reasons,

(1) It allows us to use a goal-oriented approach. It provides us a mean to examine solely the portions of our own interest in the application.

(2) In the abundant presence of polymorphism, dynamic analysis provides concise measurements in object-oriented systems.

2.4.1 Dynamic Analysis facilitates Target-oriented Comprehension

When dealing with unfamiliar software systems, the use of dynamic analysis permits to incorporate a target-oriented (or need-based) approach. When the only available knowledge about the system is of end-user functionality, it is very simple to run only those execution scenarios (from the entire list of use cases) that are directly

related to the functionality which the programmer wants to comprehend. This saves a lot of time as the execution trace is concise and to the point. Moreover the results of the analysis are better.

For example, consider the case of a programmer who wants to know how a document processor like Open Office works internally while the properties of the selected text are being changed. If we use dynamic analysis for this case, we could exercise only those scenarios that include selecting a text and then changing its properties (e.g. change the font). On the other hand, if we do not use this target-oriented approach and rely on the static analysis of the application, we must understand the whole application before we can explore the parts that are directly related to the functionality we are trying to understand.

2.4.2 Dynamic Analysis reveals actual picture of Polymorphism (Late Binding)

Polymorphism is the facility offered by modern object oriented programming languages that allows the objects of same base class but different derived classes to implement the same method of base class in different ways in the derived classes. The actual class to which the object belongs is not required to be known to the programmer in advance, hence the decision about the class of object and its behavior can be made at run-time. This introduces the concept of late binding that delays the decision about the behavior of a particular object until runtime.

The technique of polymorphism permits building programs in much efficient manner. In addition, the use of polymorphism should make the evolution of software much simpler and easier. On the other hand, the use of polymorphism can make matters worse during program comprehension process, as it turns out to be challenging to understand the exact behavior of the system, without observing the running software. This is due to the fact that one probable polymorphic method invocation is a variation-point which is able to produce a large number of different behaviors (the count of probable behaviors is obtained by adding one to the number of classes that exist in the class-hierarchy under the base class type). For example, in our first case study software system (Apache Ant), the base class Task has above one hundred

derived classes, every sub-class represents a particular command-line task that may be performed by the system.

Contrary to static analysis, when examining the software by means of dynamic analysis, the acquired vision of the software is accurate in connection with the execution scenario. The exhibited behavior belongs to the functionality that has been utilized. As a result, the number of probable variations is reduced from the superset of all theoretically possible variations to the small subset of actually executed variations during the execution-scenario.

2.4.3 Technological Support for Dynamic Analysis of Program

Profilers/Debuggers. A profiler is more often employed to examine the memory/performance parameters of an application. In contrast, a debugger is commonly utilized to wade through the running application at the programmer's desired level of detail so as to reveal the root cause for unexpected behavior.

In most cases, the built-in provisions of profiler/debugger in virtual machine environments or the operating system environments are capable of signaling the start/end of events during the execution. The programmer can easily construct a plugin for the virtual machine with the aim of being notified of these events and perform some operation during these events, for example record these events in an execution trace. The most common events that can be signaled by a profiler or debugger are the start of a call to a method/procedure, the end of a call to a method/procedure, access to data members, fields, etc.

AOP. Aspect-oriented programming (AOP) presents a novel concept for program writing, known as aspect [17]. The aspect provides the facility to implement a cross-cutting concern which does not plainly fit in to any single class or module of the application; instead it exists in many classes/modules. The source code written to implement such concerns could be confined in the advice portion of the aspect, whereas the point-cut segment of the aspect describes the points for weaving the code mentioned in the advice.

The current implementations of AOP permit the addition of a chunk of code at the start and/or at the end of a method. This facility is very helpful for writing an aspect for program tracing. Such aspect can record each method invocation or end of method call in the execution trace.

Modification of Abstract Syntax Tree. While the source code of an application is being parsed, modifications can be done in the abstract syntax tree (AST) prior to producing the AST once more in the form of regular source code. According to our information, there is no standard mechanism available for such kind of AST modification. It may be noticed that a few AOP frameworks operate in the same manner, where the weaving of aspects is done using an Abstract Syntax Tree modification process [14].

Wrapping of Methods. Method wrappers permit to capture and supplement the behavior of already available methods with the aim of wrapping additional behavior around them [16]. In our case, the supplementary behavior of tracing could be provided by wrapping of methods.

Improvised tracing. All the methods that have been stated earlier, have a planned and prepared mechanism to perform the tracing operation. Nevertheless, at times, the scope of points of interest is very restricted within the application. In such cases, improvised tracing of points of high interest can be an immediate solution.

2.4.4 The Consequences of Observing – Observer Effect

In numerous fields of pure science, observer-effect normally accounts for amendments that the observation procedure introduces in the phenomenon being observed. A conventional example of this matter arrives from the studies in quantum physics, which demonstrates the fact that observation of an electron results in the modification of its path; since the monitoring light/radiation possesses sufficient energy that can deviate the path of the electron under observation. A comparable effect has been identified in studies in social sciences, where the population under study started to deviate from their original behavior when they realized that they are being observed.

In software engineering research, the analysis of systems under observation has reported an analogous effect, known as the probe effect [21]. Utilizing dynamic analysis to observe the system, this effect could reveal itself in various manners:

- Since the software system under observation takes extra amount of time and memory during execution as compared to its normal execution time and memory; the user can probably click a button multiple times in anxiety without waiting for the system to respond. If this happens, then the execution path of the system could deviate from the normal execution scenario.
- The other possible threat whose consequences could be more severe is the impact of the observation process on thread interactions that take place in the system under observation.

As a general rule of thumb, it is advised that the overhead introduced by the extra step of observation in the running system should be as little as possible, to reduce the level of uncertainty generated by the observer effect. One possible option to minimize the overhead can be the analysis of selective portions of the program that could be important from program comprehension point of view, for example classes that are loaded early during start up of program and have strong coupling with other classes. This approach of analyzing the program during its execution (online analysis) is totally opposite to the offline (post-mortem) analysis approach in which the program is executed first and execution trace is generated. Then this execution trace is analyzed after the program has finished its execution.

2.4.5 The perils of Dynamic Analysis

The dynamic analysis is performed to capture the real picture of the system during execution, meaning that what is actually happening in the system as the program proceeds. However, there are situations where dynamic analysis could be problematic. Now we'll have a quick look into those problems and the precautionary measures taken against them.

- Most modern systems utilize the multi-threading facility to achieve some sort of parallelism. To achieve any functionality of the system typically there are several interactions among the threads in the system. In multi-processor environments, the threads can execute and interact in parallel, while the execution and coordination of threads is possible in sequential manner in uni-processor environments. The storage of all the events that happened in each thread at one common place (i.e. execution trace) can create perplexity for the programmer; because the programmer will think that the two events took place one after the other, while in actual, the events were generated by entirely different threads. This problem can be solved easily by storing the information about each active thread separately during the program execution.
- The use of class loading provisions or reflection methodology is common in many systems to enable dynamic loading of classes. The dynamic analysis mechanism based on profiler/debugger usually results in the recording of calls to the methods of the classloader. The incorrect recording of a few method calls could be possible in this case. In AspectJ, we can specify the rules for inclusion and exclusion of classes from tracing. We can easily exclude classes of least interest using AspectJ.

2.5 Use of Coupling for Program Comprehension

In this research, we aim to provide the programmer with a small number of preliminary points, which can be utilized to begin the tracking of structural dependencies in order to gain familiarity with the system. These preliminary points will be identified using coupling.

2.5.1 What is Coupling?

Constantine et al presented the concept of coupling in the form of a heuristic for improved module design [22]. Constantine defined coupling as the measurement of strength of a bond developed by a connection from a particular module to other

modules. Since the definition provided by Contantine is a bit casual, so we will adhere to the definition of coupling in [9] which states that "*two things are coupled if and only if at least one of them 'acts upon' the other. X is said to act upon Y if the history of Y is affected by X, where history is defined as the chronologically ordered states that a thing traverses in time*".

2.5.2 Why dynamic coupling metrics?

For quite a time, research has been carried out on the subject of coupling measurements, for example in the perspective of quality metrics. Mostly the coupling metrics have been estimated statically through the source code (or any other static model of program structure) by analyzing the dependencies that exist among various program elements. The precision of coupling metrics that are estimated statically declines quickly in the presence of dead code, dynamic binding and inheritance. This loss of precision has compelled for the search of more accurate alternatives like dynamic coupling measures, a research area in software engineering that is developing [10]. The following definition of dynamic coupling measures has been used by us:

To define dynamic coupling measures, one needs to perform the analysis of interactions that take place between objects at runtime. If an object acts on another object, then the two objects are said to be dynamically coupled. When it is evident from execution trace that there exists a relationship of method call between the two objects p and q (provided that the method call was originated from p), we say that Object p has acted on Object q. Moreover, dynamic coupling exists between two classes provided that one or more object(s) of one of these classes act upon the other class object(s) at runtime.

2.5.3 Possible variations for calculating dynamic coupling metrics

There are various means to measure dynamic coupling. Based on the application context in which these measures are selected for use, the rationale behind each measure can be presented [10]. The variations of dynamic coupling measures defined in [10] are presented in Table 2.

Table 2: Variations of Dynamic coupling measures [10]

Entity	Granularity (Aggregation Level)	Scope (Include/Exclude)	Direction
Object	Object Class (set of) Use case(s) System	Library objects Framework objects Exceptional use cases	Import/Export
Class	Class Inheritance Hierarchy (set of) Subsystem(s) System	Library classes Framework classes	Import/Export

Variation of Entity of Measurement. As measurement of dynamic coupling is based on the runtime information about the system, the coupling can be computed at the ***object*** as well as the ***class level*** [10].

Variation of Strength Level of Coupling. The coupling strength levels quantify the amount of association between the entities of measurement (objects or classes). The quantification of the association between the entities of measurement (objects or classes) may be done at the following levels [10]:

1) ***Counting Dynamic Messages*** *(All Method Invocations)*. The run-time information of a program can be used to determine how many times each method (of object/class) is invoked by the other method (of another object/class). This count can then be used to quantify the strength of the association between the entities of measurement.
2) ***Counting Distinct Method Invocations***. Another simpler option to quantify the strength of the association is to determine the number of unique method calls between the entities of measurement. It must be noticed that each distinct method is counted only once which is dissimilar from counting all method invocations (that may include recurrence of method call).
3) ***Counting Distinct Classes***. For a given object/class, count the number of distinct objects/classes to which it is associated (coupled); utilizing the run-time information of a program.

Variation of Aggregation Level. The aggregation of dynamic coupling measures can be made at various levels of detail. In the case of dynamic object coupling, the calculations can be made by means of objects, at the same time the aggregate coupling at the class level can be computed by adding the coupling of all the objects of that class. Various sorts of aggregates can be obtained based on the measurement level. Possible aggregation levels include system level, sub-system(s) level, package level etc [10].

Variation of Search Space. For calculation of dynamic coupling metrics, we can make variations in the search space by ***including/excluding classes*** in the search space. For instance, the classes that are linked as a library can be excluded occasionally if they are not worth investigating [10].

Variation of Direction of Coupling (Import/Export). Suppose we have a class c_1 whose method m_1 calls the method m_2 of class c_2. We can interpret this coupling relation between the classes c_1 and c_2 in terms of client-server relation. The class c_1 is the *client class* that **imports** the services of *server class* c_2 (which **exports** its services for its clients). This is the base idea of ***import*** and ***export*** coupling [10].

On the basis of possible variations for calculating dynamic coupling measures (described in section 2.5.3), a total of 12 dynamic coupling metrics have been defined by [10]. We list only 6 (out of 12) dynamic coupling metrics in Table 3.

Table 3: Dynamic Coupling Metrics [10]

Coupling Direction	Measurement Entity	Strength Level	Metric Name
Import **C**oupling	**O**bject	**D**ynamic messages	IC_OD
		Distinct **M**ethod	IC_OM
		Distinct **C**lasses	IC_OC
	Class	**D**ynamic messages	IC_CD
		Distinct **M**ethod	IC_CM
		Distinct **C**lasses	IC_CC

2.5.4 How the dynamic coupling metrics are calculated

In Table 3, the names of 6 (out of 12) dynamic coupling metrics from [10] are presented. Three dynamic coupling metrics namely IC_CM, IC_CC and IC_CC′ have been referred in the Chapter 6 (Results and Evaluation). We'll now explain the calculation of these two metrics using a simple example.

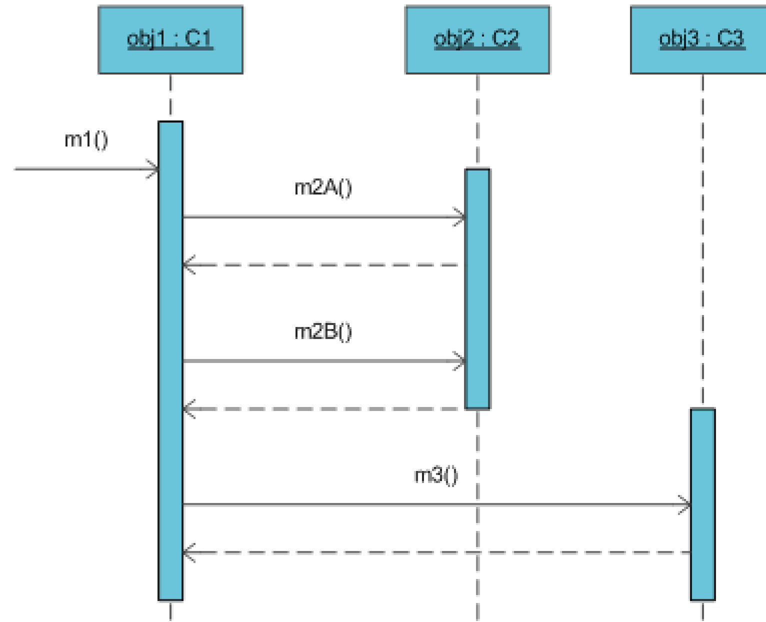


Figure 1: A Simple Sequence Diagram

Figure 1 shows a simple sequence diagram that involves 3 classes namely C1, C2 and C3. The method m1() of class C1 first calls the method m2A() of class C2. After the control is returned to method m1(), it next calls the method m2B() of class C2. Finally method m3() of class C3 is called by the method m1() of class C1. The dynamic coupling metrics IC_CM, IC_CC and IC_CC′ [12] for class C1 are calculated as follows:

Table 4: Calculating Dynamic Coupling Metrics – Simple Sequence Diagram

Metric Name	Calculation	Score
IC_CM(c1)	{(m1,c1,m2A,c2), (m1,c1,m2B,c2), (m1,c1,m3,c3)}	3
IC_CC(c1)	{(m1,c1,c2) ,(m1,c1,c3)}	2

IC_CC′(c1)	{(m2A,c1,c2), (m2B,c1, c2), (m3,c1, c3)}	3

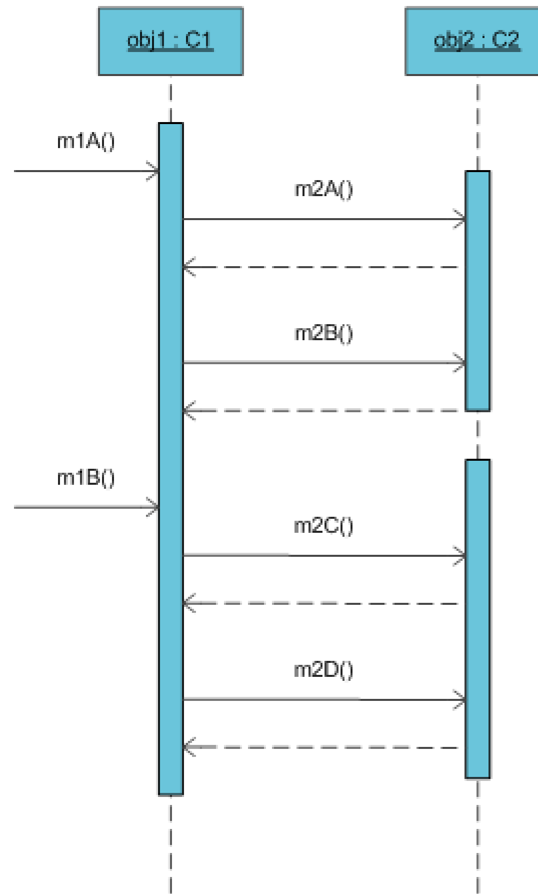


Figure 2: Another Simple Sequence Diagram

Figure 2 shows another simple sequence diagram that involves 2 classes namely C1 and C2. The method m1A() of class C1 first calls the method m2A() of class C2. After the control is returned to method m1A(), it next calls the method m2B() of class C2. Now the method m1B() of class C1 is invoked which invokes methods m2C() and m2D() of Class C2. The dynamic coupling metrics IC_CM, IC_CC and IC_CC′ [12] for class C1 are calculated as follows:

Table 5: Calculating Dynamic Coupling Metrics – Only Two Classes Involved

Metric Name	**Calculation**	**Score**
IC_CM(c1)	{ (m1A,c1,m2A,c2), (m1A,c1,m2B,c2), (m1B,c1,m2C,c2), (m1B,c1,m2D,c2) }	4

IC_CC(c1)	{ (m1A,c1,c2) , (m1B,c1,c2) }	2
IC_CC′(c1)	{ (m2A,c1,c2) , (m2B,c1, c2) , (m2C,c1, c2) , (m2D,c1, c2) }	4

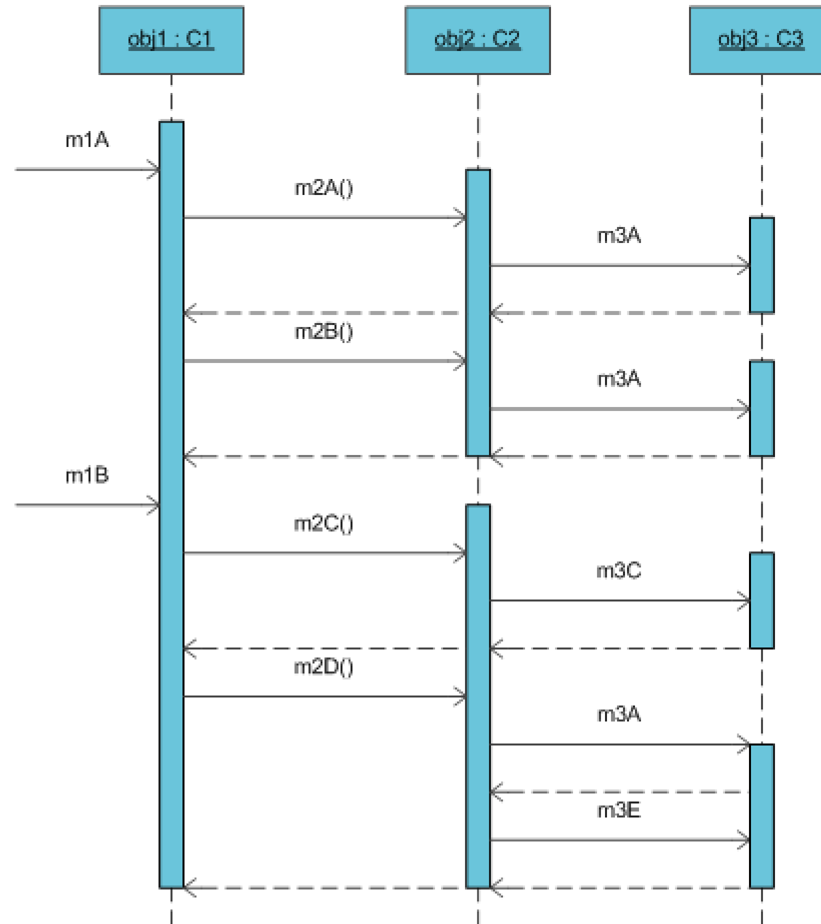


Figure 3: Slightly Complicated Sequence Diagram

Figure 3 shows a slightly complicated sequence diagram that involves 3 classes namely C1, C2 and C3. The method m1A() of class C1 first calls the method m2A() of class C2, which further calls the method m3A() of class C3. After the control is returned to method m1A(), it next calls the method m2B() of class C2, which further calls the method m3A() of class C3. Now the method m1B() of class C1 is invoked which invokes methods m2C() of Class C2, which further calls method m3C() of Class C3. Next method m2D() of Class C2 calls the methods m3A() and m3E() of Class C3.

The dynamic coupling metrics IC_CM, IC_CC and IC_CC′ [12] for classes C1 and C2 are calculated as follows:

Table 6: Calculating Dynamic Coupling Metrics – Complex Sequence Diagram

Metric Name	**Calculation**	**Score**
IC_CM(c1)	{ (m1A,c1,m2A,c2), (m1A,c1,m2B,c2), (m1B,c1,m2C,c2), (m1B,c1,m2D,c2) }	4
IC_CC (c1)	{ (m1A,c1,c2) , (m1B,c1,c2) }	2
IC_CC′(c1)	{ (m2A,c1,c2) , (m2B,c1, c2) , (m2C,c1, c2) , (m2D,c1, c2) }	4
	For Class C2	
IC_CM(c2)	{ (m2A,c2,m3A,c3), (m2B,c2,m3A,c3), (m2C,c2,m3C,c3), (m2D,c2,m3A,c3), (m2D,c2,m3E,c3) }	5
IC_CC (c2)	{ (m2A,c2,c3), (m2B,c2,c3), (m2C,c2,c3), (m2D,c2,c3) }	4
IC_CC′(c2)	{ (m3A,c2,c3), (m3C,c2,c3), (m3E,c2,c3) }	3

2.6 Key Classes of Software

Usually a software system is composed of many programming constructs including components, classes, modules etc. These basic elements of programming work together to achieve a particular functionality and their teamwork produce the idea of coupling. Highly coupled modules are interconnected with large number of other modules; as a result considerable amount of effort is required to gain an understanding of such modules or to perform any type of maintenance of these modules. The basic desire to achieve low level of coupling between modules originates from this observation. However, it is evident that a certain level of coupling will be present in the system all the time because modules/classes have to work in cooperation with each other to achieve a particular functionality. The classes responsible for facilitating the cooperation of work (among various classes of the system) tend to have an administering role in the application and usually possess a high level of coupling. An analogous concept, known as key classes was used by the author in [8]:

"The key concepts of most of the object-oriented systems are implemented by few classes which are known as key classes. These few classes can be identified easily as they exhibit some common characteristics. Usually they possess a coordinating role in the application and hence issue orders to huge number of other classes and utilize them for implementing the functionality. As a result, there exists a tight coupling between the key classes and the rest of the system. Moreover, they have a propensity of being complex, as they realize a great deal of the functionality of the system."

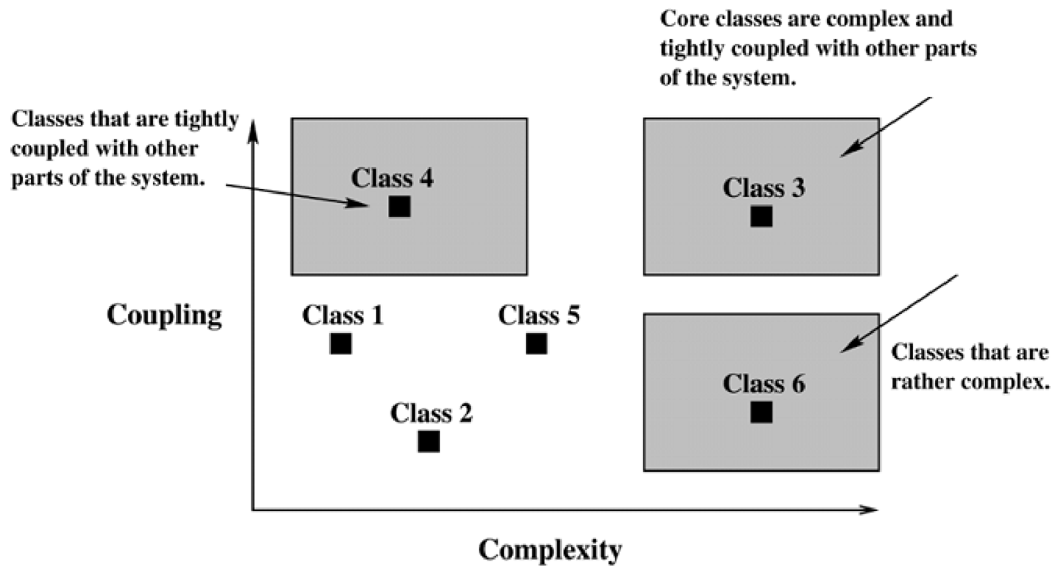


Figure 4: Key Classes of Object-Oriented Software [8]

2.6.1 Existing Approaches to Identify Key Classes of Software

Authors in [23] used the combination of Weighted Methods per Class (WMC) and Data Abstraction Coupling (DAC) metrics to identify key classes. WMC represents the total of the complexity score of all methods in class. DAC counts those attributes of the class whose type is defined by other classes. The authors compared the identified key classes with original developers' opinion.

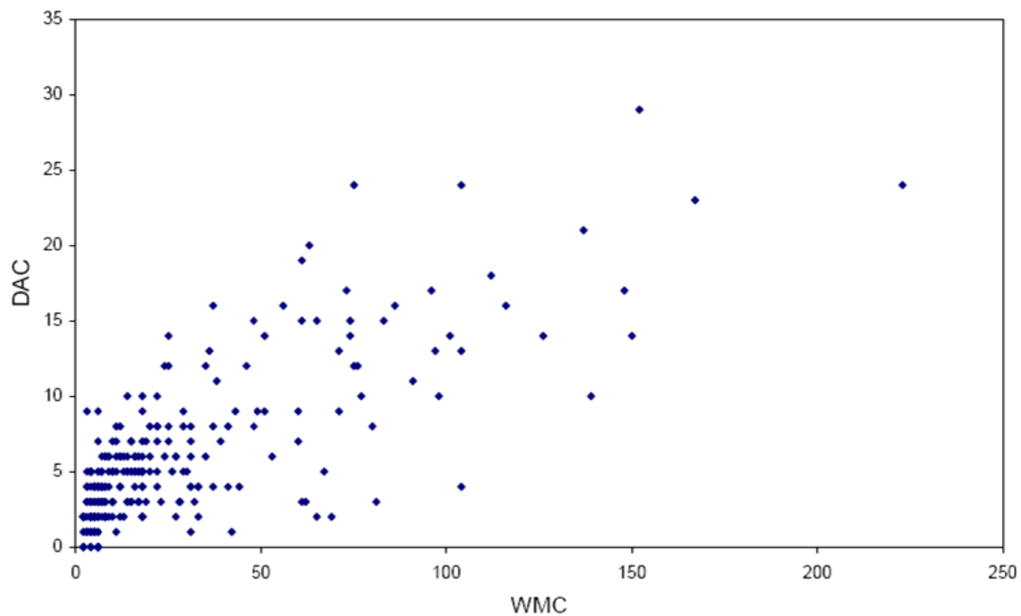


Figure 5: Using WMC and DAC to identify Key Classes [23]

Authors in [24] proposed the extraction of compacted call graph from the execution trace of the system. They applied the HITS web mining algorithm [25] on compacted call graph for identifying important classes in a system's architecture.

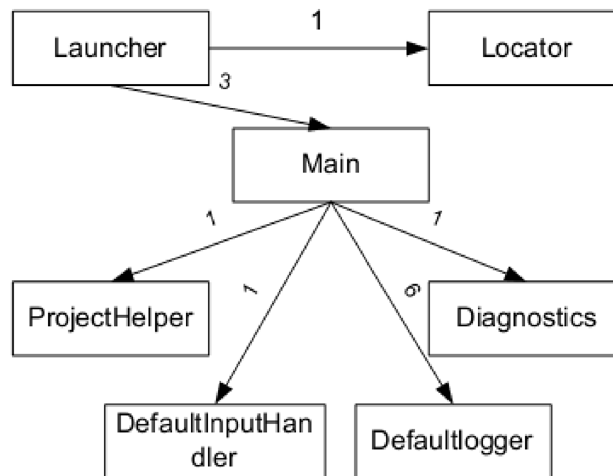


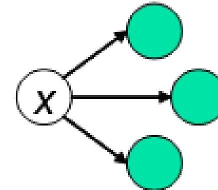
Figure 6: A Compacted Call Graph [24]

Authors of [14] proposed to apply HITS web mining algorithm [25] on static structure of the application to find key classes of application. They used 4 static

coupling metrics from [13] and compared the results with the design documentation. They found that the precision and recall was considerably low with the use of static coupling metrics.

Hub Scores h(p):

- hub scores are updated with the sum of all authority weights of pages it points to



Authority Scores a(p):

- authority scores are updated with the sum of all hub weights that point to it

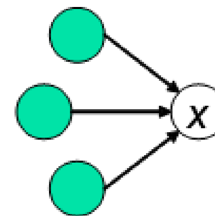


Figure 7: HITS Webmining Algorithm [25]

Authors of [12] proposed a variant of dynamic coupling metric presented by [10] to identify key classes of application under study. Their proposed variant of dynamic coupling metric also considered indirect coupling. They compared the results with original developers' opinion.

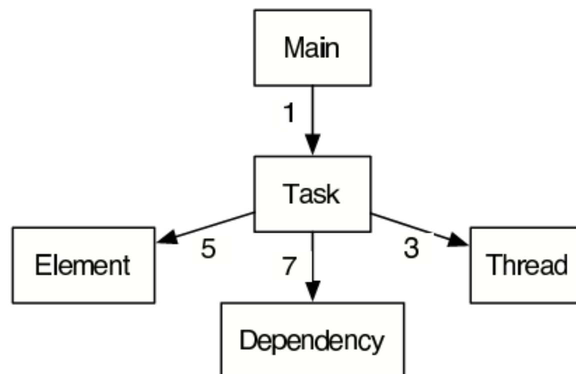


Figure 8: Indirect Coupling example [12]

Table 7: Summary of existing approaches to identify Key Classes of Software

S#	Authors	Approach	Drawbacks
1	Bauer et al., 1999	Static analysis of source code using static coupling metrics (WMC and DAC)	Low precision in the presence of polymorphism
2	Zaidman et al., 2004	Post execution analysis of execution trace, extraction of Compacted Call Graph from execution trace, Applying HITZ webmining algorithm on this Compacted Call Graph	Scalability issues due to huge size of execution trace
3	Zaidman et al., 2006	Calculating coupling scores of classes using Static Analysis of source code, Creating a Graph of classes using these coupling scores, Applying HITZ webmining algorithm on this Graph.	Low precision in the presence of polymorphism
4	Zaidman et al., 2008	Post execution Analysis of execution trace, Calculation of Dynamic Coupling Metric IC_CC' for each class from execution trace, Creating a Graph of classes using IC_CC' as score, applying HITZ webmining algorithm on this Graph	Scalability issues due to huge size of execution trace

CHAPTER 3

OVERVIEW OF TECHNOLOGIES USED IN THIS RESEARCH

3.1 Introduction to the Technology Used For Dynamic Analysis

3.1.1 An overview of AspectJ

AspectJ is an implementation of Aspect Oriented Programming (AOP) in Java as well as a language specification. The constructs to support aspect oriented concepts and their semantics are defined by the language specification. The compilation, debugging and other tools for documentation of code are offered by the language implementation.

The language constructs in AspectJ are extension of the Java programming language, hence a Java Program is an AspectJ program as well. The class files produced by the AspectJ compiler are compliant with the specification of Java byte code; hence the class files generated by the AspectJ compiler can be interpreted by any compliant JVM. Since the Java language is available at the base level of AspectJ, all the provisions in Java are accessible to the programmers, making the use of AspectJ easier for them.

One of the strongest points of AspectJ has been the availability of effective tool support. The job of a compiler is done by the aspect weaver. Other useful tools include the debugger which is aspect-aware, a source code documentation generator and a tool for visualizing the effect of advice on various parts of the system. In addition, the AspectJ is now integrated with majority of the eminent IDEs for Java developers (for instance, Oracle JDeveloper, Netbeans, eclipse etc.) that makes AspectJ a useful implementation of Aspect Oriented Programming for developers working with Java language.

3.1.2 What are Joinpoints?

The concept of Joinpoints is central to AspectJ; they are defined as unambiguous, definite points in the execution of program. Prospective joinpoints could be before/after a method call, checking of a condition, the start of a loop or before/after assigning the value to a variable. Every joinpoint posseses an associated context, for instance, a joinpoint before call to a method provides access to the target object and the arguments passed to the method by using context information.

During the execution of program, any uniquely recognizable point can become a joinpoint. But AspectJ has placed some restrictions on joinpoints and only those joinpoints are available that can be utilized in an organized manner. The following pointcuts can be utilized in AspectJ:

- Call/execution of a method
- Call/execution of a Constructor
- Accessing the variable for reading/writing
- Execution of exception handling routine
- Initialization of class/object

3.1.3 The role of Pointcuts

These are the programming constructs used to select joinpoints, infact they provide a mean for specifying a group of joinpoints. Another important role of pointcuts is in exposing the context (of joinpoint) to the implementation of advice.

Pointcuts for call to constructors/methods: These pointcuts interrupt the execution once the arguments of the method are evaluated, but prior to calling the method in actual. These pointcuts are declared using the syntax like call(Method/ConstructorDecleration). These pointcuts are illustrated with the help of examples in Table 8.

Table 8: Pointcuts for call to constructors/methods

Syntax of Pointcut	Description
call(protected String SomeClass.Function1())	Call to Function1() of SomeClass that takes

	no argument, returns a String, having protected access
call(* void SomeClass.Function1(..))	Call to Function1() of SomeClass that takes any argument, returns void, having any access modifiers
call(* SomeClass.Function1(..))	Call to Function1() of SomeClass that takes any argument and returns any type
call(* SomeClass.Function1*(..))	Call to any method whose name starts with Function1 in SomeClass
call(* SomeClass.Function1*(Integer,..))	Call to any method whose name starts with Function1 in SomeClass but the data type of first parameter is Integer
call(* *.Function1(..))	Call to Function1() that belongs to any class in default package
call(SomeClass.new())	Call to no arguments constructor of SomeClass
call(SomeClass.new(..))	Call to any arguments constructor of SomeClass
call(SomeClass+.new(..))	Call to any arguments constructor of SomeClass or its subclass. (wildcard + indicates the use of subclass)
call(public * org.someorg..*.*(..))	All the public methods of classes that are declared within the root package of org.someorg

Pointcuts for execution of constructors/methods These pointcuts intercept the execution of constructors/methods. Opposite to the pointcuts for call, the body of constructor/method is represented by pointcuts for execution. These pointcuts are declared using the syntax of the form execution(Method/ConstructorDecleration).

Table 9: Poincuts for execution of constructors/methods

Pointcut	**Description**

execution(public void SomeClass.Function1())	Execution of Function1() in SomeClass that takes no argument, returns nothing, having public access
execution(void SomeClass.Function1(..))	Execution of Function1() in SomeClass that takes any no. and type of arguments, return type is void, and any access modifiers
execution(* SomeClass.Function1(..))	Execution of Function1() in SomeClass that takes any argument and returns object of any type
execution(* SomeClass.Function1*(..))	Execution of any method in SomeClass whose name starts with Function1
execution(* SomeClass.Function1*(int,..))	Execution of any method in SomeClass whose name starts with Function1 and the first argument is of int type
execution(* *.Function1(..))	Execution of Function1() that belongs to any class in default package
execution(SomeClass.new())	Execution of no arguments constructor of SomeClass
execution(SomeClass.new(..))	Execution of any arguments constructor of SomeClass
execution(SomeClass+.new(..))	Execution of any arguments constructor of SomeClass or one of its subclass. (wildcard + indicates the use of subclass)
execution(public * org.someorg..*.*(..))	All the public methods that are declared within the root package of org.someorg

Pointcuts for access of field The access to a field of the class for reading/writing is captured by these pointcuts. For instance, every access to the field named out defined in the class System (may be through the use of the statement System.out.print) can be captured using such kind of pointcuts. Whether the access of field is for reading or for writing purpose, you can capture both types of access. For instance, the access of field xField in SomeClass for the purpose of writing by using a

statement like SomeClass.xField = 12 can be captured easily. The pointcut that captures the access for reading is normally outlined as *get(FieldNamePattern)*, whereas the pointcut that captures the access for writing takes the shape like *set(FieldNamePattern)*. Wildcards can be utilized in FieldNamePattern, the way they have been used in the call/execution pointcuts described above.

Pointcuts for handlers of Exception The specified type of exception handlers can be captured during execution using these pointcuts. Normally they use the notation like *handler(TypeOfException)Pattern*.

Pointcuts for Class-initialization The specified type of static-class initialization code, that is usually specified in the class definitions using static blocks, can be captured during its execution through these pointcuts. Normally they are represented by the notation like *staticInitializePattern*.

Pointcuts based on Lexical-structure Every joinpoint that lie within the lexical structure of the method or a class can be captured by such pointcuts. The *within(TypePattern)* notation is used to describe the pointcut that captures the code which comes under the lexical structure of a class (it also includes the inner class). The *withincode(ConstructorMethodPattern)* notation is used to describe the pointcut that captures the code which comes under the lexical structor of constructor or method (it also includes the local classes).

Pointcuts based on Control-flow These poincuts are used to capture the control flow of other pointcuts, that is how the program instructions flow during the execution. For instance, during the execution of program, if method m1 passes the control to method m2 (in fact by calling m2); then method m2 is said to be in the control flow of method m1. The call to every method, the access to every field, even the exception handlers executed as a result of invoking a particular method can be captured using this kind of pointcut. The *cflow(pointcut)* notation is used to describe the pointcut capturing the flow of control for additional pointcuts (including itself). The *cflowbelow(pointcut)* notation serves the similar purpose except that the pointcut itself is not included.

Pointcuts for type of argument, target and self The pointcuts are used for capturing the joinpoints on the basis of the type of argument, the object at the target and the self-object. The context at the joinpoint can be captured by these contructs only. The notation of *this(ObjectIdentifier/TypePattern)* is used to describe the pointcuts on the basis of self object, whereas the syntax of the form *target(ObjectIdentifier/TypePattern)* is used to describe the pointcuts on the basis of target object. Finally the pointcuts for type of the argument are represented by the syntax like *args(ObjectIdentifier/TypePattern, …).*

Pointcuts that are named If the name for the pointcut is specified explicitly, then it is known as named pointcut. These pointcuts can be reutilized for overriding a pointcut, in the definition of other pointcuts, definition of a portion of advice and so on.

Pointcuts having no name (Anonymous) These pointcuts are defined at the point where they are used, in a manner similar to anonymous class definition. Usually they are used as a part of definition of another pointcut (or specification of an advice). The reuse of anonymous pointcuts is not possible just as anonymous classes could not be resused.

Gathering context information **–** Access of data in the joinpoint is often required by the advice implementation. For instance, a certain type of information about the method and its arguments, commonly known as context, might be required by the advice to perform a logging operation. The gathering/exposing of context related information at the execution point is therefore provided by the pointcuts, which is then passed to advice implementation. The pointcuts of args(), this() and target() are offered by AspectJ to expose the context.

Reflection **–** Reflection is supported by AspectJ in a limited form. The information at the execution point of any pointcut can be examined using reflection supported by AspectJ. A special object named *thisJoinPoint* is available in the body of each advice and the joinpoint related information is encapsulated by this object. The availability of this reflective information is vital for implementation of debugging aspect (or logging aspect) for any application.

3.1.4 Advices

The code to be executed when program execution reaches the particular pointcut is specified by advice. Three options are provided by AspectJ for linking the advice to joinpoint that are before the execution, after the execution or around the execution of joinpoint. The advice defined using the keyword *before* is executed prior to the joinpoint, whereas the advice defined using the keyword *after* is executed just past the joinpoint. There is a provision in *after* advice to specify whether it should be executed after returning normally(without exception) or after an exception has been thrown or after both cases (normal and exception). The advice defined using the keyword *around* basically surrounds the joinpoint and has the authority to decide whether the execution of joinpoint should be continued. The decision to carry on with a changed set of arguments can be made while using around advice.

Here is an example of before and after advice that displays thisJoinPoint and recent time before and after the call of all public methods in MyClass:

```
before () : call(public * MyClass.*(..)) {
      System.out.println ("Before: " + thisJoinPoint + " " +
System.currentTimeMillis  ());
}
after () : call(public * MyClass.*(..)) {
      System.out.println ("After: " + thisJoinPoint + " " +
System.currentTimeMillis  ());
}
```

Figure 9: Simple Advice example before/after call of public method

Every call to connection.close() method is captured by the advice in following example. If the connection pooling has been enabled, the connection is put back to the pool; else the execution is advanced with the proceed() method. The context information provided by target() is also utilized in the advice:

```
void around(Connection conn) : call(Connection.close()) && target(conn) {
  if (enablePooling) {
     connectionPool.put(conn);
  } else {
     proceed();
  }
}
```

Figure 10: Advice example that uses context collected by target

3.1.5 The role of Aspects

The role of Aspects in AspectJ is similar to the role of classes in Java; they are the basic unit of modularization. The pointcuts and advices are encapsulated in an aspect. There are some similarities between aspects and classes, e.g. an aspect may possess data members and functions, inherit properties/functions from other aspects or classes, and can provide implementation of interfaces. On the other hand, a major difference between a class and an aspect is that an instance for an aspect cannot be created using the new operator.

Classes are allowed to declare pointcuts using AspectJ. Static pointcuts must be declared inside a class. On the other hand, the advices are not allowed to be declared inside class, they must be specified within aspects.

Any aspect and pointcut can be declared as abstract. The pointcut declared abstract act like the abstract method of a class: it allows you to delay the decisions to the extending aspects. A derived aspect that extends an abstract aspect can offer the actual implementation of abstract pointcuts.

3.1.6 What is Load-Time Weaving (LTW)?

In principal, AspectJ is a programming language. It provides the support for primitive Java types and a novel construct known as aspect. One option of using AspectJ is writing a program in its language and then compiling the program using its compiler that will generate standard byte-code (from source files) which can be run

easily by JVM. There is also a provision in AspectJ that a pre-compiled jar file of aspects can be weaved into your application classes. This weaving of application classes can be performed as an extra step during the build process or even at runtime when the class is being loaded by the virtual machine. The weaving of application classes by the AspectJ agent during load time is known as Load-time weaving (LTW).

A pre-compiled aspect library can be used in an easy and very flexible manner during development with the aid of load-time weaving. The –javaagent option available since version 5 of JDK is the simplest way to utilize LTW.

Using aop.xml to configure Load-time Weaving – The configuration of AspectJ agent for load-time weaving is done by aop.xml file(s) that must be placed within the search path of class loader. Every aop.xml file includes a listing of aspects utilized for weaving application classes, type patterns indicating types to be considered for weaving, and other initial settings to configure the weaver. There are two main sections in the aop.xml file: one or more aspects for the weaving process are defined in the *aspects* section. The inclusion/exclusion of aspects in the weaving process is controlled from here. The types to be woven and supplementary weaver options are defined in the *weaver* section.

The easiest manner to declare the aspect (for its use by the weaver) is to mention the aspect type (in a fully qualified manner) in the aspect element of aop.xml file. Aspects can be declared and defined inline using the aop.xml file to configure the weaver. *Concrete-aspect* element is used to define such inline aspects. If the inline aspect extends some abstract aspect, then an implementation must be provided for the abstract poincuts that have been inherited. With this useful mechanism, the configuration of auxiliary infrastructure aspects can be easily externalized to cater the situations where the pointcut definition itself is part of the service configuration.

There can be multiple ***include*** and ***exclude*** elements defined in the aspect tag. (The default implementation uses all the defined aspects for weaving). The aspects that will be utilized by the weaver can be restricted by specifying include or exclude elements. The aspects qualifying the include pattern are utilized, where as the aspects qualifying the exclude pattern are ignored during weaving.

It may be noted that all aspects are affected by ***include*** and ***exclude*** elements, regardless of the fact that they are defined in the same aop.xml file or a different aop.xml file. After the application of this filtering mechanism, in case an aspect has not been declared, a lint warning is generated to help preventing unanticipated behavior.

The specification of the types that need to be woven and the passing of parameters to the weaver is done using the ***weaver*** element. The weaver weaves all the visible types in the absence of any include element specification. The byte-code of classes can be saved to disk using the ***dump*** element both before (in case of runtime weaving) and after the weaving process to help diagnosing the problem.

CHAPTER 4

RESEARCH METHODOLOGY

Up till now, we have discussed the theoretical foundations of our approach to discover the key classes that can be quite useful in the early stages of program comprehension. This chapter first provides an introduction to the research methodology used by us. Next we present an overview of the open source case study that we have used in the experiment to compare the performance of our proposed solution with the other approaches.

4.1 Research Methods

The study of software engineering on the basis of observations and experiences is known as *Empirical* software engineering. In empirical software engineering researchers try to ascertain a scientific approach for the given problem of software engineering.

An important role is played by the empirical studies within the software engineering research. An empirical study can be carried out in many ways, depending upon the research project. Following research methods are widely used in the field of software engineering for empirical studies:

4.1.1 Conducting a Survey

A survey is commonly used to capture the broader picture of the happenings over a large group of projects. Hence, the survey approach can help in the evaluation of a software technology over a large scale. The survey based approach has the benefit that it can confirm/reject the results of research by generalizing a large number of projects, using statistical analysis approach.

4.1.2 Controlled Experimentation

In a formal experiment, the factors that affect the phenomenon under investigation need to be controlled. For instance, suppose we desire to investigate that whether programs written in language *A* (say C++) results in better quality code than the programs written in language *B*. In this case, we need to make sure that the factor of skill level of the programmers (the subjects of experiment) is equal for both types of programmers.

4.1.3 Case Study

Case study based approach outshines other research methods when it comes to gaining an insight to a complex problem. This approach can be used to extend the past experiences or refine the knowledge obtained by previous research. A thorough contextual analysis of a few circumstances and their association is emphasized in the case study. It facilitates deep understanding of a particular case or problem, thus enabling the analysis of many variables and capturing the reality in detail. The potential of a case study to obtain detailed knowledge makes it a strong explorative mechanism to establish new research as well.

The research method of case study has been used by researchers in diverse disciplines for many years. It has been widely used in the field of computer science/software engineering. In fact, the historical data demonstrates that it has been the most common empirical validation model in the field of empirical software engineering. An important application of case studies is the industrial evaluation of tools and methods of software engineering.

Since we are proposing an extension to the previous research on dynamic coupling metric and we want to compare our proposed approach with the previous research accomplished in the past, so the case study based approach best suites our needs. Hence we have used the case study based approach in our research.

4.2 The organization of the experimental system

4.2.1 Selection criteria for the case study

Two open source software system were adopted for the experiments conducted during this research. These systems were selected as a case study based on the following two factors that made these software systems specifically attractive for the evaluation of our approach to aid initial program comprehension:

- The open access and free availability of software facilitates the repeatability of the same or analogous experiments in future research projects.
- The existence of detailed design documentation is extremely helpful to validate the results of experiments in program comprehension. In addition, the free availability of this detailed design documentation has an added advantage of ensuring repeatability.

Finally, Apache Ant 1.6.1 and Jakarta JMeter 2.0.1 were selected by us since they match closely with the criteria mentioned above. Critics may propagate that nearly all open source systems will posses these properties, but our vote for this specific case study was also driven by the fact that that this software system has been used in other analogous experiments, which provided us with a baseline for our experiment and made the comparative analysis easier for us.

4.2.2 Selection of Use Case for Tracing

The selection of the use case for tracing and calculation of metric is a necessary pre-requisite for dynamic analysis. At one side, a usage scenario covering major functionalities of the system can be advantageous in the reverse engineering process of huge systems. Alternatively, a scenario that only covers the functionality that is of major concern to the reverse engineer can be useful to minimize the size of solution set (which is a ranked list of classes in our case, sorted by their importance level). Consequently, it permits the use of target based approach to fully concentrate on the focal point during program comprehension efforts. In our research context, we suggest

to select use case(s) covering major functionalities of the system to increase the chances of detecting majority of the key classes of the system.

4.2.3 The baseline for the experiment

The availability of detailed design documentation and the results of other analogous experiments in [11, 12] performed on the same software systems(Apache Ant and Jakarta JMeter) provided us with a baseline to be used in our experiment. The baseline includes the classes identified by the original authors of the code and/or the developers currently maintaining the system as must-to-comprehend classes in order to perform any type of maintenance of the system. Nevertheless, this baseline is still an approximation as it reflects the point of view of highly skilled programmers that may be entirely different from the opinion of a greenhorn maintainer struggling to get a grasp on the system.

The major advantage of this baseline is that it allows us to perform an intrinsic assessment of our approach. By intrinsic assessment we mean that we have used the point of view of original developers of the system for comparison with the results acquired by our approach.

4.2.4 Outcome

The outcome of our proposed heuristic is an ordered list of classes arranged in descending order with respect to their significance. We have opted to present the top 15% classes to the developer from the entire list of classes because of the following reasons:

- The design documents have brought this thing into our knowledge that nearly 10% of the system classes must be comprehended prior to perform any significant modification in the system. We have gained an additional margin of 5% because we are using a heuristic.
- Because of the cognitive constraints, the volume of data provided to the developer to start his comprehension efforts should be minimal. The developer should not be overloaded with information.

- The third reason was that by raising the margin to 20%, we observed a very minor increase in recall. On the other hand, the precision fell significantly.

4.3 Overview of the 1st Case Study -- Apache Ant

4.3.1 What is Apache Ant?

Apache Ant is an eminent tool that has been mostly utilized for building software projects on Java platform. Ant runs as a single thread. There is no graphical user interface for its execution because it is invoked through command line interface. A large number of external libraries are utilized by Ant (XML library by Apache Xerces is just one example), however its own footprint is comparatively small. It is quite flexible and can be extended by the user. It is based on XML and the build files for the projects have a specific XML format.

Regardless of the fact that it is an open source product, Ant is being utilized in open source as well as industrial systems. Besides, it has also been made available in various Integrated Development Environments (e.g. Netbeans, JDeveloper, eclipse etc.) designed to work with Java Platform. The basic distribution of Ant has been extended in various manners and one of its flavors known as nANT has been ported to the .NET platform as well.

We have used the version 1.6.1 of Apache Ant that ships with the source code. There are 1216 classes in this version of Ant which are written in Java. The number of classes specific to Ant is 403 that have nearly 83 KLOC. The distribution contains a large number of classes that belong to the frameworks or libraries used by Ant for supporting tasks such as manipulation of regular expression (Apache ORO) and XML parsing (Apache Xerces).

4.3.2 Use Case for Tracing

In our experiment with Apache Ant, we selected the use case of building the source project of Ant using its own binary release. One hundred and twenty seven (127) classes were involved in this scenario. Keeping in view that a total of 403 classes

constitute the Ant build, the figure of 127 appears to be quite small in the initial glimpse. This gap in the expected and actual number of classes that took part in the scenario can be explained by the existence of some extraordinarily broad and deep inheritance hierarchies in the architecture of Ant. For instance, there are 104 subclasses that directly inherit the Task class. Every individual subclass is responsible for handling a particular type of command line task for example *mkdir*, *rmdir*, etc. Since majority of usage scenarios do not exercise all of the available commands (the usage of some commands is not even possible at one time e.g. the commands implemented for heterogeneous platforms or version management systems), hence it can be safely said that our scenario (covering 127 classes) exercises the major portion of the functionality offered by the Ant system.

The following two basic reasons were behind our choice of this specific usage scenario:

- A fair portion of features are practiced in this scenario. In addition, it incorporates the build commands that are used in most cases such as make directory, delete directory, copy folders/files, creating a jar file and so on.
- It is very easy to perform and/or repeat this scenario with the help of build.xml file that is present in every source release of Ant.

4.3.3 Architecture of Ant

In this section, we will discuss the role played by the five important classes (that are deemed significant by original developers) during the execution of the build file. The information about these classes has been extracted from the detailed design documentation that is available freely:

1. **Project**: The instance of Project class is created as soon as Ant is started by the Main class. The parsing of build.xml file is done by the Project instance with the assistance of supplementary objects. Targets and Elements are contained in the xml file for build.
2. **Target**: All the targets described using the build.xml files are instantiated as objects of this class. When the parsing of build.xml is finished, the build model

is composed an instance of project that may have numerous targets or at least one target that serves as default target for events at upper level.

3. **UnknownElement**: Every parsed element is temporarily placed in objects of class UnknownElement. In the course of parsing, the instances of class UnknownElement are stored along with the specific Target (which is related to them) in a data structure similar to tree. At the end of the phase of parsing, all dependencies are resolved. At this time, the method named makeObject() belonging to this class is called. The correct type of object is instantiated by this method according to the values of data placed in the objects of this class.
4. **RuntimeConfigurable**: There exists a RuntimeConfigurable instance for each corresponding UnknownElement instance. The information regarding the configuration of element is contained in it. The instances of class RuntimeConfigurable are also stored along with the specific Target (which is related to them) in a data structure similar to tree.
5. **Task** It is the parent class for UnknownElement. It also serves as a base class for every task instantiated by using the method named makeObject() of class UnknownElement.

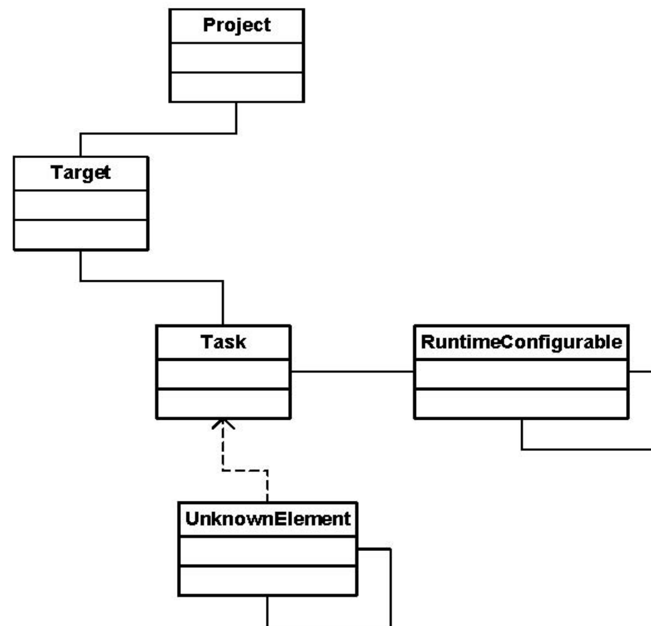


Figure 11: Class diagram showing 5 important classes of Apache Ant

We have made an effort to capture the relationship between the five aforementioned classes in Figure 11. The complete list of classes included in the baseline for Ant case study, extracted from the detailed design documentation is as follows:

1. Project
2. Target
3. Task
4. RuntimeConfigurable
5. UnknownElement
6. Introspection Helper
7. ProjectHelper2
8. ProjectHelperImpl
9. ElementHandler
10. Main

4.4 Overview of the 2nd Case Study – Jakarta JMeter

4.4.1 What is Jakarta JMeter?

Jakarta JMeter 2.0.1 is a java-based desktop application built to test the web applications. It can be utilized for functional verification of software system; also it helps to perform load-testing (i.e. to assess performance of the application). Mainly, it is utilized for testing web-based applications, however, it is capable of handling SQL queries as well that are made through JBDC. In addition, the flexible architecture allows plugins to be created for additional protocols. Results of performance measuring can be presented in a variety of graphs, while results of the functional testing are simple text files with output similar to output from regression tests.

JMeter is a desktop application which posseses a rich GUI to exercise the features, threads are utilized in abundance and majority of the functionality is supplied by the standard Java API (e.g. web-related functionality). The source code of version

2.0.1 of Jakarta JMeter comprises of nearly 700 classes, whereas the core part of Jakarta JMeter software is made of 490 classes (23 KLOC).

4.4.2 Use Case for Tracing

The use case that was exercised for tracing in this case study consists of testing a HyperText Transfer Protocol (HTTP) connection to localhost (JBoss Server) hosting Ant documentation pages. Actually, we organized JMeter application in such a way that it tested the above HTTP connection one-hundred times and at the end the results were visualized in a simple graph. Without the tracing operation enabled, the time taken by the use case was 82 seconds. It is worth mentioning here that several threads were instantiated to initiate 100 HTTP connections so that concurrent access by multiple users can be simulated for the web application. Hence the JMeter case study serves as an instance of multi-threaded software system.

4.4.3 Architecture of JMeter

Now we will discuss the tasks performed by the key classes (that are judged by original authors of code as key classes) of JMeter.

The TestPlanGUI, as evident from its name is a user-interface component. It allows the addition and customization of tests. JMeterGUIComponent class holds each test that has been added. When the creation of TestPlan has been finished by the end user, the JMeterGUIComponents contain this information which is then placed in TestElement classes for further processing.

The objects of TestElement class are saved in JMeterTreeModel which is a tree datastructure. The JMeterEngine now takes this tree datastructure and instantiate JMeterThread(s) for every test with the assistance of TestCompiler. The ThreadGroups are used to form logical grouping of JMeterThreads. In addition, a TestListener is instantiated for each test which is used to catch the results of JMeterThread(s).

The complete list of classes included in the baseline for JMeter case study, extracted from the detailed design documentation is as follows:

1. AbstractAction
2. JMeterEngine
3. JMeterTreeModel
4. JMeterThread
5. JMeterGuiComponent
6. PreCompiler
7. Sampler
8. SampleResult
9. TestCompiler
10. TestElement
11. TestListener
12. TestPlan
13. TestPlanGui
14. ThreadGroup

CHAPTER 5

THE PROPOSED APPROACH

In this chapter, first we describe the problems with the existing approaches to identify key classes. Then we have described the basic idea and the implementation of our proposed approach to discover key classes of software. The introduction to the technology (AspectJ) that we have used to implement our approach is already provided in Chapter 3. Here we explain the algorithm used to implement our approach. The basic idea for resolving the problem of key class identification and the rationale behind the use of such idea is described as well.

5.1 Problems with Existing Approaches to Identify Key Classes

The approaches based on static coupling metrics have the disadvantage that they lose precision in the presence of polymorphism in object-oriented system [14].

The approaches based on dynamic coupling metrics suffer from scalability issues due to the huge size of the execution trace [14].

- The **size** of the execution trace was 2 GB in the case study of Ant
- The **time** to calculate the metric from this 2 GB of data was 45 minutes
- The **I/O overhead** of the tracing operation was very high
  - e.g. execution of Ant took 23 seconds *without tracing*.
  - whereas execution of Ant took just under one hour *with tracing*.

Our main goal in this research is to devise an efficient approach to identify key classes that could overcome the scalability issues introduced by the existing approaches based on dynamic coupling metrics. It is evident that the main cause of the scalability issues is the offline analysis approach or the so called post-mortem analysis (i.e. running the program, generating the execution trace and then extracting the coupling information from those large text-based files whose size is in gigabytes). So we propose the **online analysis approach** in which the dynamic coupling information is calculated in parallel to the execution of the program using AspectJ.

5.2 Basic Idea behind our Approach

It has already been mentioned that we are searching for key classes that posses a central place in the architecture of the system. Our expectation is that these classes have a supervisory role in the application, giving instructions to large number of classes and dictating them the work to perform. These classes will be requesting the other classes for their services, which imply that the key classes will be tightly coupled with other classes.

5.2.1 Hypothesis

Our hypothesis is that the key classes posses a coordinating role in the application as they manage and request services of large number of other classes, so they must be loaded in the application prior to other classes from which they request services. So the sequence in which the classes are loaded in the application must also be taken into consideration (in addition to coupling information) to identify the key classes.

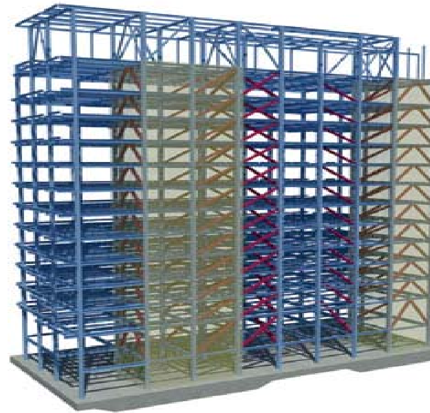


Figure 12: Core Structure Analogy from Civil Engineering

We have built an analogy from civil engineering structures like bridge/buildings that the core part of the structure is laid first and the rest of the structure is based on that core part. So we proposed that the classes loaded during the start up that have strong coupling with other classes are most likely to be key classes and are prime candidates for early program comprehension.

The existing dynamic coupling metrics do not take into account the loading order of the class in application. So we have to extend the definition of existing dynamic coupling metric for this purpose. In [10] a total of twelve metrics have been defined by authors. We have used the definition of one metric named *IC_CM* (explained in Section 2.5.3) from their work and extended this metric to meet our objective of finding the key classes of the system according to loading order of the classes. We will now provide the definition of the metric from [10] and then discuss our proposed extension.

Table 10: Dynamic coupling measures [10]

C	Set of classes in the system.
M	Set of methods in the system.
$\mathbf{R_{MC}}$	$R_{MC} \subseteq M \times C$ Refers to methods being defined in classes.
IV	$IV \subseteq M \times C \times M \times C$ The set of possible method invocations.
$\mathbf{IC_CM(c_1)}$	$\#\{(m_1, c_1, m_2, c_2) \mid (\exists (m_1, c_1), (m_2, c_2) \in R_{MC}) \land c_1 \neq c_2 \land (m_1, c_1, m_2, c_2) \in IV\}$
Our Extended Version of IC_CM	
$\mathbf{IC_CMS(c_1)}$	$\#\{(m_1, c_1, \mathbf{lo}, m_2, c_2) \mid (\exists (m_1, c_1), (m_2, c_2) \in R_{MC}, \mathbf{lo \in N}) \land c_1 \neq c_2 \land (m_1, c_1, m_2, c_2) \in IV\}$

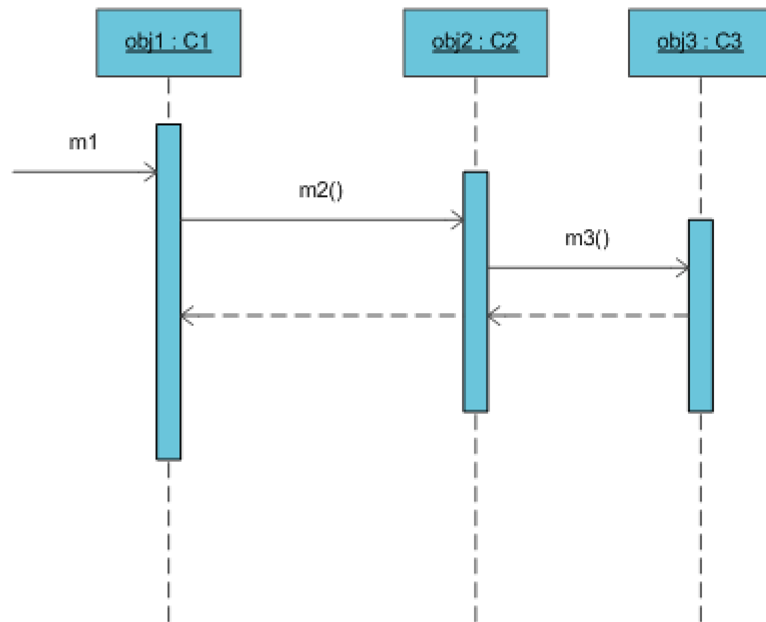


Figure 13: Sample Sequence Diagram to calculate IC_CMS

Figure 13 shows a simple sequence diagram that involves 3 classes namely C1, C2 and C3. The method m1() of class C1 first calls the method m2() of class C2. Method m2() next calls the method m3() of class C3. It is obvious that the object of class C1 is loaded 1st, object of class C2 is loaded 2nd while object of class C3 is loaded 3rd. The dynamic coupling metrics IC_CM and IC_CMS for classes C1 and C2 are calculated as follows:

Table 11: Calculating IC_CM and our extended version of IC_CM

Metric Name	**Calculation**	**Score**
IC_CM(c1)	{(m1,c1,m2,c2)}	1
IC_CM(c2)	{(m2,c2,m3,c3)}	1
IC_CMS(c1)	{(m1,c1, **0**, m2,c2)}	1
IC_CMS(c2)	{(m2,c2, **1**, m3,c3)}	1

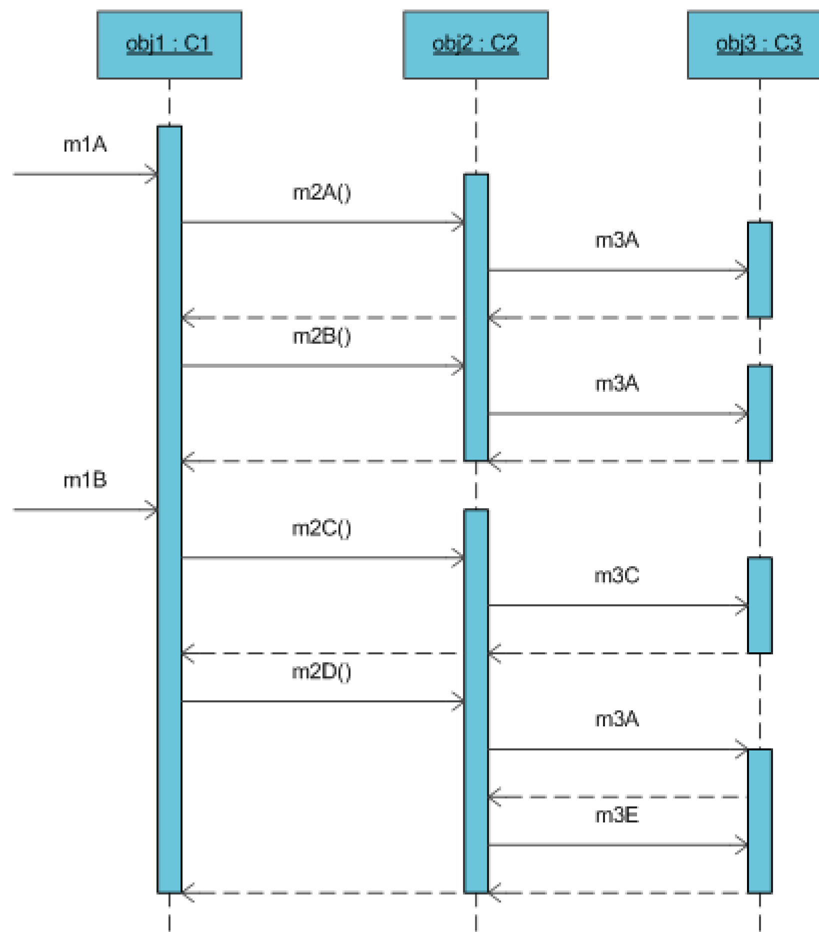


Figure 14: Slightly Complicated Sequence Diagram to Calculate IC_CMS

Figure 14 shows a slightly complicated sequence diagram that involves 3 classes namely C1, C2 and C3. The method m1A() of class C1 first calls the method m2A() of class C2, which further calls the method m3A() of class C3. After the control is returned to method m1A(), it next calls the method m2B() of class C2, which further calls the method m3A() of class C3. Now the method m1B() of class C1 is invoked which invokes methods m2C() of Class C2, which further calls method m3C() of Class C3. Next method m2D() of Class C2 calls the methods m3A() and m3E() of Class C3. The dynamic coupling metrics IC_CM, IC_CC, IC_CC′ [12] and IC_CMS for classes C1 and C2 are calculated as follows:

Table 12: Calculating Dynamic Coupling Metrics

Metric Name	Calculation	Score
IC_CM(c1)	{ (m1A,c1,m2A,c2), (m1A,c1,m2B,c2), (m1B,c1,m2C,c2), (m1B,c1,m2D,c2) }	4
IC_CC (c1)	{ (m1A,c1,c2) , (m1B,c1,c2) }	2
IC_CC′(c1)	{ (m2A,c1,c2) , (m2B,c1, c2) , (m2C,c1, c2) , (m2D,c1, c2) }	4
IC_CMS(c1)	{ (m1A,c1, **0**, m2A,c2), (m1A,c1, **0**, m2B,c2), (m1B,c1, **0**, m2C,c2), (m1B,c1, **0**, m2D,c2) }	4
For Class C2		
IC_CM(c2)	{ (m2A,c2,m3A,c3), (m2B,c2,m3A,c3), (m2C,c2,m3C,c3), (m2D,c2,m3A,c3), (m2D,c2,m3E,c3) }	5
IC_CC (c2)	{ (m2A,c2,c3), (m2B,c2,c3), (m2C,c2,c3), (m2D,c2,c3) }	4
IC_CC′(c2)	{ (m3A,c2,c3), (m3C,c2,c3), (m3E,c2,c3) }	3
IC_CMS(c2)	{ (m2A,c2, **1**, m3A,c3), (m2B,c2, **1**, m3A,c3), (m2C,c2, **1**, m3C,c3), (m2D,c2, **1**, m3A,c3), (m2D,c2, **1**, m3E,c3) }	5

Our variant of IC_CM. We have made a variation of the IC_CM metric with the inclusion of a new numeric parameter that is initialized with a value of zero and is incremented each time an object of a distinct class is initialized. We name this parameter as "**loading_order**". Whenever a constructor of any class is called during the execution of program, we first check if the constructor of this class is being called for the first time? If yes, we say that this class is being loaded for the first time in the application so we add this class to the list of distinct classes that have already been loaded. The numeric parameter (loading_order) is incremented by one and is stored along with the other information like source/target of method/class.

It should be noted that although the scores for classes C1 and C2 is same (i.e. 1), for IC_CM and IC_CMS. But our extended version of IC_CM gives priority to class C1, since it is loaded prior to class C2 in the application. Whereas the original version of IC_CM fails to note this distinction and treats classes C1 and C2 at equal level because of equal coupling score.

5.3 Implementation

The technology discussed in Chapter 3 is used in the implementation of the approach described in this thesis. In proposed methodology, our variant of the dynamic coupling metric is calculated during program execution using the method interception facilities provided by AspectJ. Our approach uses the concept of *distinct method invocations* i.e. the coupling between two classes c_1 and c_2 that exists because of a method m_1 of c_1 calling the method m_2 of c_2 is recorded only once during the entire execution scenario. If method m_1 of c_1 calls the method m_2 of c_2 many times, it is not recorded after each call because we have already captured that a run-time coupling exists between the two classes (c_1 and c_2). Due to this processing time overhead can be reduced by many times. Moreover, the extra cycle of reading large execution trace file (as done in offline analysis) is avoided that saves a major portion of the time spent on calculations.

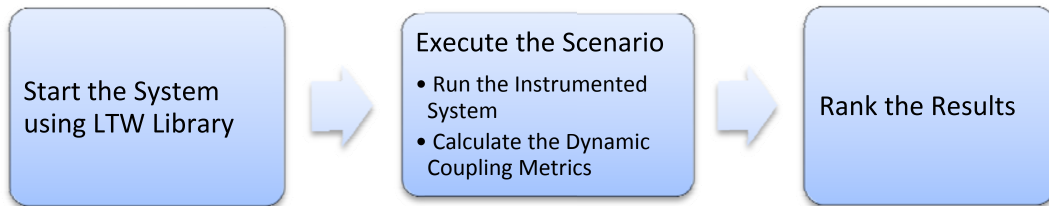


Figure 15: Overview of the Approach

5.3.1 Start the System using AspectJ Library

We have used AspectJ for dynamic analysis of the system [45]. AspectJ supports load-time weaving of applications. While using load-time weaving, the application is treated as a black-box, since any application can be monitored/traced using AspectJ library without modifying the application code. This implies that our technique is non-invasive; the source code of the application is not a pre-requisite for our approach. Instead the technique can be applied to the deployed application without modifying/re-compiling the source code. The only requirement of our approach is the addition of our

custom built aspect library to the application class-path and the launch of the system with load-time aspect weaving enabled.

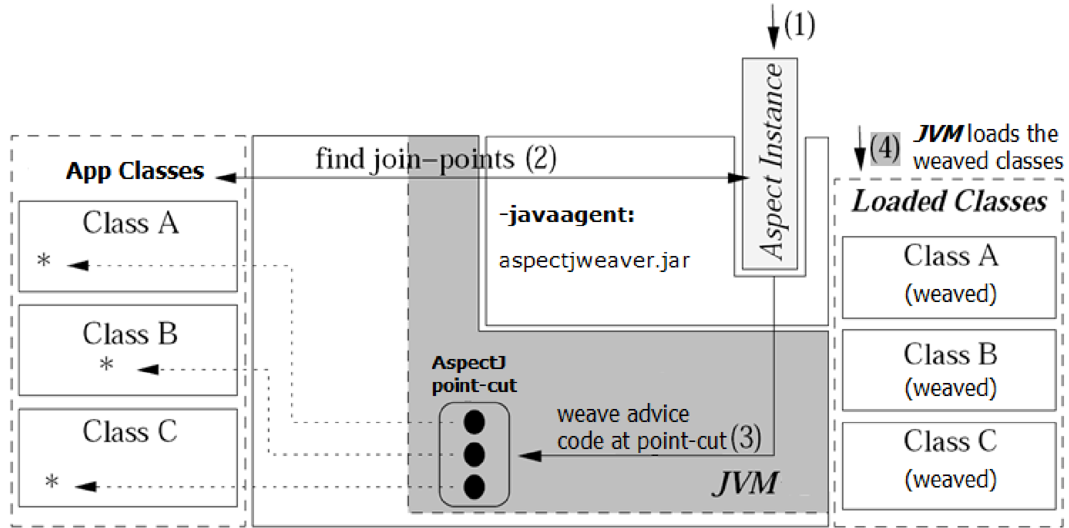


Figure 16: Load-time weaving of Application Classes using aspectjweaver.jar

The above figure explains the load-time weaving process for application classes. The -javaagent option of the JVM is utilized to specify that the application classes will be transformed by the specified aspect library using the aspectjweaver.jar. The Aspect Instance finds the join-points in the application classes that are to be loaded by the JVM. Next the advice code is weaved at the specified point-cuts in the application classes. In the final step, the JVM loads the modified version of the application classes (weaved classes) that will now execute according to the advice.

5.3.2 Run the Instrumented System and Calculate Metrics

When we run the instrumented System, AspectJ allows us to intercept each and every method call and provides us with the relevant information like the name of the class, name of the method, method signature, line number of code etc. [45]. System classes and class libraries can be excluded easily to filter unwanted classes. It can also be specified that only specified classes of the application should be intercepted. Inclusion/exclusion of classes is at the programmer’s discretion. We have built our approach around on-line analysis of the application, since we calculate the metrics in

parallel to program execution. This is opposite to off-line analysis approach in which the system is executed first and after the execution is finished the execution traces are analyzed to extract the information. As the system gets executed, we record relevant information in a linked list like data-structure.

5.3.2.1 Algorithm for Calculation of Dynamic Coupling Metric

1. Start with the entry point of the application (e.g. main method).
2. Assign an initial value of 0 to the ***"loading_order"*** variable.
3. Initialize **loaded_classes_list** with zero size list.
4. Before the execution of any method/constructor, do the following:
    a. Get the name of the previously called method/class from stack_list.
    b. Get the name of the currently called method/class.
    c. Record the coupling relationship between previously called method/class (m_1/c_1) and the currently called method/class (m_2, c_2) in the form of (m_1, c_1, m_2, c_2, **loading_order**). The repetition is not considered, only the distinct combinations of (m_1, c_1, m_2, c_2, **loading_order**) are recorded.
    d. Check if this class is present in the **loaded_classes_list**?
    If No,
        i. Assign the value of "loading_order" variable to this class.
        ii. Add this class to the loaded_classes_list.
        iii. Increment the value of loading_order variable.
        iv. Add this method to the invoked_methods_list of this class.
    If Yes,
        i. Add this method to the invoked_methods_list of this class (if not already added)
5. Add the currently method/class to the stack_list.
6. Repeat the steps 4 and 5 till the end of the application.
7. At the end of the application you will have the following:

a. The order in which the classes are loaded by the application in **loaded_classes_list**.
b. Each class will have the list of methods that are invoked during the execution scenario in **invoked_methods_list** of that class.
c. The coupling relationship between classes in the form of (m_1, c_1, m_2, c_2, **loading_order**). It is again emphasized that the repetition is not considered; only the distinct combinations of (m_1, c_1, m_2, c_2, **loading_order**) are recorded.

5.3.2.2 Flow of activities for calculation of Dynamic Coupling Metric

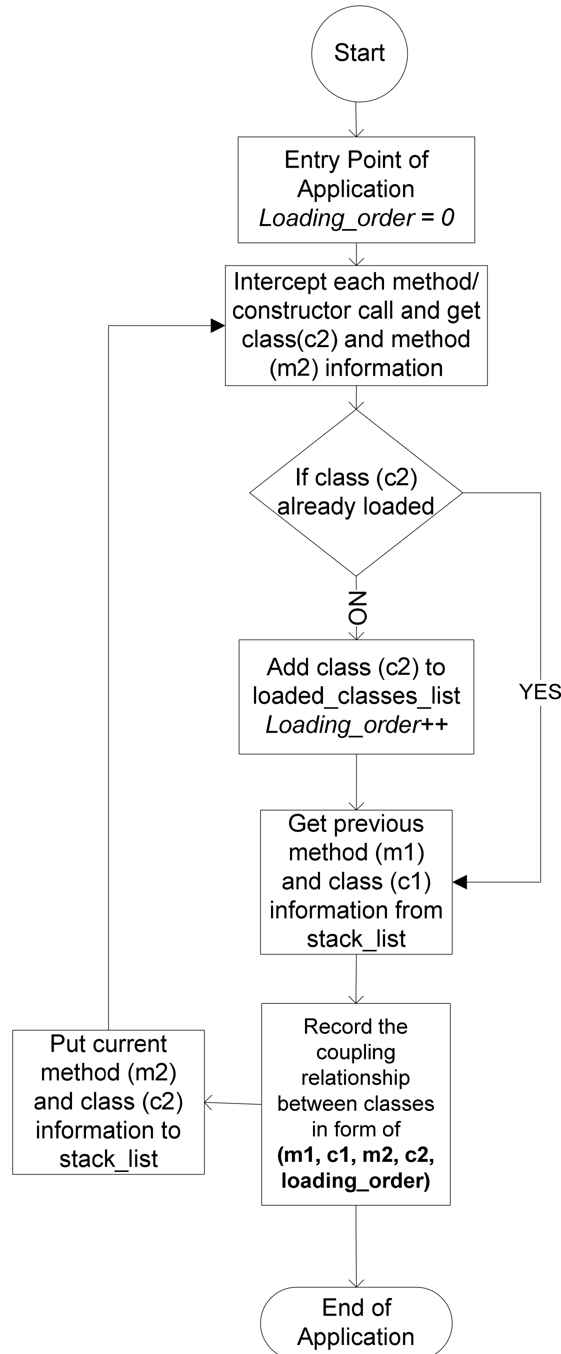


Figure 17: Flow of activities for calculation of dynamic coupling metric

5.3.3 Rank the Results

At the end of the execution, we rank the classes according to their metric value (i.e. classes are sorted on the **loading_order** ascending and count of invoked methods of other classes descending) and display the top fifteen percent results to the user [45].

5.3.4 System Requirements

System prototype is developed using Eclipse IDE and java development kit 6. For running this software prototype there is a requirement of Java Runtime Environment 6 and database handling is done using Oracle 10g which must be installed and database should be configured for proper running of this software. In tabular form, the specification of the system on which experiment was conducted is given in Table 13.

Table 13: System Specifications

System Processor	AMD Athlon Dual-Core QL-62 2.0 GHz
Hard Disk	150 GB
RAM	4 GB
Operating System	Windows 2000 Server, Windows 2003 Server, Windows XP, Windows 7
Runtime Environment	Java Runtime Environment 6
Database Server	Oracle 10g
Case Study Softwares	Apache Ant 1.6.1 Jakarta JMeter 2.0.1
AspectJ Version	AspectJ 1.6.7
IDE	Eclipse Galileo

5.4 How the proposed approach is evaluated and validated

We have performed an intrinsic evaluation of our approach using open source software as case study (details in Chapter 6). Following three major evaluation criteria have been used to evaluate the approach:

- The recall of the approach (or retrieval power of the approach).
- The precision of the approach.
- The time consumed in the complete process.

Similarly, the validation of the proposed approach is performed with the help of precision and recall. In addition, we have computed the amount of effort in our approach which could help in figuring out the return on investment in terms of time.

5.5 Practical use of our approach

In order to obtain the benefits of our approach, the programmer working on the new software can use the following steps:

- Choose a major use case of the system.
- Start the system using our AspectJ library.
- Run the selected use case of the system.
- Our heuristic will determine and present the key classes to the programmer as a ranked list.
- The programmer can use these classes as a starting point for his comprehension process.

CHAPTER 6

RESULTS AND EVALUATION

In this chapter results of the developed prototype system are evaluated against the existing approaches. For this purpose the evaluation criteria is identified and our approach is compared with the results reported by the experiments conducted on the same software system.

6.1 Results of 1st Case Study – Apache Ant

To evaluate the accuracy of our heuristic, we have used as baseline the most important classes, provided by [11, 12] (extracted from design documents of Apache Ant 1.6.1 and Jakarta JMeter 2.0.1) in their experiment of finding key classes of a software system.

The top 15% classes identified by our approach are presented in Table 14. The 15% mark is set by the authors in [12]; they argue that the documentation of Apache Ant mentions that nearly 10% of the total classes are deemed important by the original authors of Apache Ant. Hence the 15% mark is quite reasonable for the evaluation of the approach.

There are 19 classes that have been marked as key classes by our approach. Out of these 19 classes, 9 classes were identified correctly as shown in the Table 14 on next page. Whereas 10 classes are false positives.

Table 14: Classes identified by our approach (Top 15%) – Apache Ant

Our identified classes	Baseline Classes [11,12]
Project	Project
UnknownElement	UnknownElement
AntTypeDefinition	
ComponentHelper	
Main	Main
ElementHandler	ElementHandler
IntrospectionHelper	IntrospectionHelper
Property	
FileUtils	
ProjectHelper2	
PropertyHelper	
DefaultLogger	
ProjectHelper	ProjectHelper
ProjectHandler	
RuntimeConfigurable	RuntimeConfigurable
TargetHandler	
RootHandler	
Task	Task
Target	Target
	TaskContainer

6.2 Evaluation of 1st Case Study – Apache Ant

6.2.1 Criteria Used for Evaluation

Following are the main criteria on which proposed approach is evaluated and compared to existing techniques:

i. Precision

Precision is the proportion of classes that lies within the baseline in all the discovered classes.

$$Precision = \frac{\text{Number of relevant classes retrieved}}{\text{Number of retrieved classes}} \quad \textbf{(1)}$$

ii. Recall

Recall is the fraction of the classes, which are relevant to the baseline, that are successfully retrieved.

$$Recall = \frac{Number\ of\ relevant\ classes\ retrieved}{Number\ of\ relevant\ classes} \quad \textbf{(2)}$$

iii. Fallout

It is the proportion of all non-relevant classes present in the retrieved classes.

$$Fallout = \frac{Number\ of\ non{-}relevant\ classes\ retrieved}{Number\ of\ retrieved\ classes} \quad \textbf{(3)}$$

iv. F Measure

This is the weighted harmonic mean of precision and recall. It trades off between precision and recall.

$$F = \frac{(\beta^2+1)PR}{\beta^2 P+R} \quad \textbf{(4)}$$

where —— **(5)**

- Where F is F-measure, P is precision and R is recall
- The default well adjusted F-measure that fairly weights precision and recall uses the parameters:

– or **(6)**

v. The effort (Time) it takes to perform the complete analysis, from start to finish.

6.2.2 Precision Comparison

Precision of our approach is compared with other approaches in the following Table.

Table 15: Comparison of Precision % -- Apache Ant

Metric Name	IC_CM	IC_CC	IC_CC'	IC_CM + HITS	IC_CC + HITS	IC_CC' + HITS	Our Approach
Precision %	21	37	37	32	42	47	47

Graphical form of the above table is given in Figure 18.

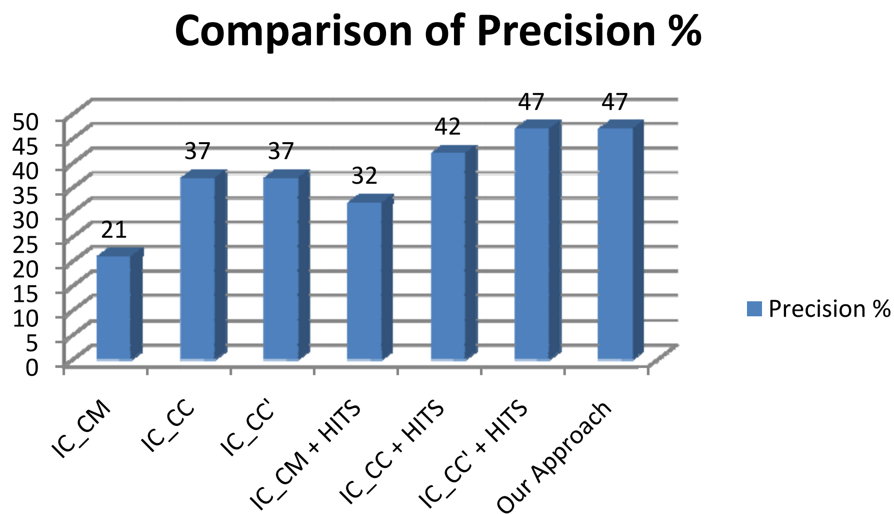


Figure 18: Comparison of Precision % -- Apache Ant

6.2.3 Recall Comparison

Recall of our approach is compared with existing approaches in the following Table.

Table 16: Comparison of Recall % -- Apache Ant

Metric Name	IC_CM	IC_CC	IC_CC'	IC_CM + HITS	IC_CC + HITS	IC_CC' + HITS	Our Approach
Recall %	40	70	70	60	80	90	90

Graphical form of the above table is given in Figure 19.

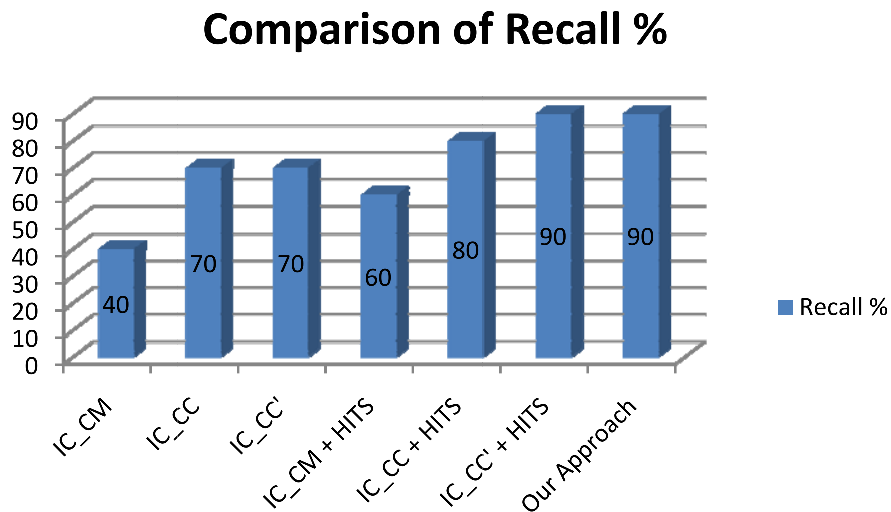


Figure 19: Comparison of Recall % -- Apache Ant

6.2.4 Fallout Comparison

Fallout of our approach is compared with existing approaches in the following Table.

Table 17: Comparison of Fallout % -- Apache Ant

Metric Name	IC_CM	IC_CC	IC_CC'	IC_CM + HITS	IC_CC + HITS	IC_CC' + HITS	Our Approach
Fallout %	79	63	63	68	58	53	53

Graphical form of the above table is given in Figure 20.

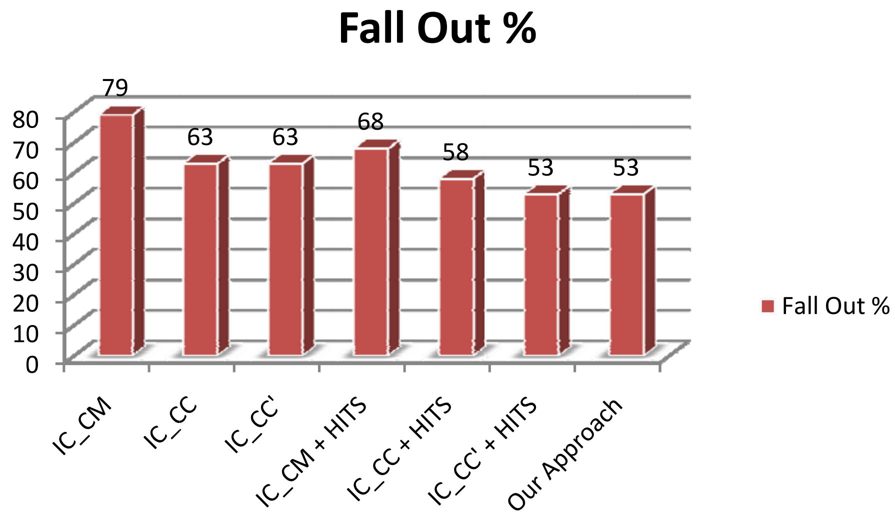


Figure 20: Comparison of Fallout % -- Apache Ant

6.2.5 F-measure Comparison

F-measure of our approach is compared with other approaches in the following Table.

Table 18: Comparison of F-Measure % -- Apache Ant

Metric Name	IC_CM	IC_CC	IC_CC'	IC_CM + HITS	IC_CC + HITS	IC_CC' + HITS	Our Approach
F-Measure	27	48	48	41	55	61	61

Graphical form of the above table is given in Figure 21.

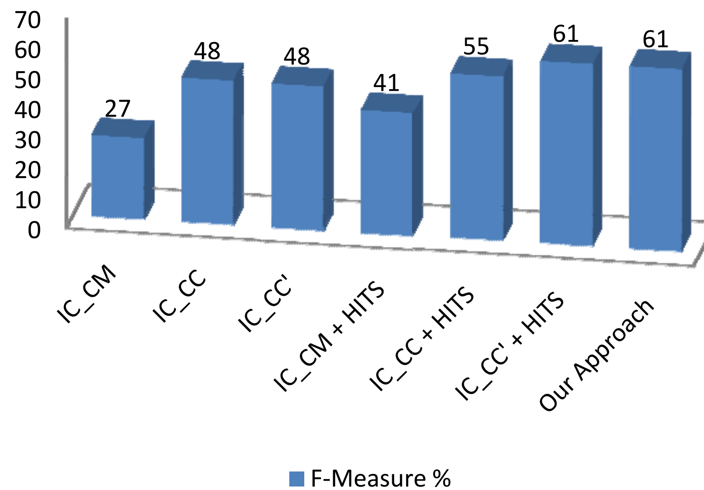


Figure 21: Comparison of F-Measure % -- Apache Ant

6.2.6 Time Usage – (effort to perform the complete analysis, from start to finish)

In this criteria time is measured for the normal execution time taken by the machine to build the Ant source project without instrumentation as well as the time taken by the machine to build the same Ant source project with the overhead of instrumentation involved in our approach. The time for all existing approaches was taken from the published results of the experiments conducted by the respective author. Time measurement is taken as a relative measure, not as an absolute measure. For instance, the results of previous experiments reported that the original execution time of Ant build was 23 seconds; but with overhead involved in measuring the dynamic coupling information, the execution time elevated to 60 minutes (3600 seconds). It is evident that the existing approach has increased the original execution time by a factor of 156. With our approach the original execution time of Ant build was 26 seconds; but with overhead involved in measuring the dynamic coupling information, the execution time elevated to 10 minutes (600 seconds). It is noticeable that our approach has increased the original execution time by a factor of 23 which is far more efficient than the factor of 156. On the basis of this principle all of the measurements are taken and their results shows that new technique presented in this thesis is far better than the existing one in term of time usage.

Table 19: Details of results for Time Usage – Apache Ant

APPROACH	ORIGINAL EXECUTION TIME (SECONDS) *(A)*	EXECUTION TIME WITH APPROACH (SECONDS) *(B)*	OVERHEAD FACTOR *(B/A)*
IC_CM	23	6300	273
IC_CC	23	6300	273
IC_CC`	23	6300	273
IC_CM + HITS	23	6300	273
IC_CC + HITS	23	6300	273
IC_CC` + HITS	23	6300	273
Our Approach	15	600	40

Graphical form of the above table is given in Figure 22.

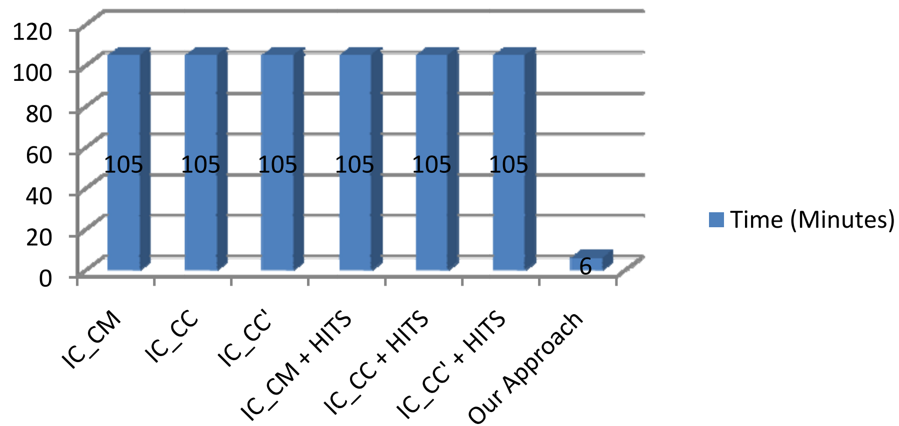


Figure 22: Comparison of time usage of approaches – Apache Ant

6.3 Results of 2nd Case Study – Jakarta JMeter

The top 15% classes identified by our approach in the experiment with Jakarta JMeter are presented in Table 20.

Table 20: Classes identified by our approach (Top 15%) – Jakarta JMeter

Our identified classes	Baseline Classes [11,12]
JMeter	
SaveService	
JMeterProperty	
JMeterTreeModel	JMeterTreeModel
TestElement	TestElement
AbstractTestElement	
Arguments	
LoopController	
HTTPSampler	
HTTPSamplerBase	
SampleResult	SampleResult
TestPlanGui	TestPlanGui
ArgumentsPanel	
JMeterEngine	JMeterEngine
TestPlan	TestPlan
ValueReplacer	
Command	
GuiPackage	
MainFrame	
TestListener	TestListener
TestCompiler	TestCompiler
ListenerNotifier	
ThreadGroup	ThreadGroup
JMeterGUIComponent	JMeterGUIComponent
MenuFactory	
JMeterThread	JMeterThread
SamplePackage	
Sampler	Sampler
	AbstractAction
	PreCompiler

There are 28 classes that have been marked as key classes by our approach. Out of these 28 classes, 12 classes were identified correctly as shown in the Table 20. Whereas 16 classes are false positives.

6.4 Evaluation of 2nd Case Study – Jakarta JMeter

6.4.1 Precision Comparison

Precision of our approach is compared with other approaches in the following Table.

Table 21: Comparison of Precision % -- Jakarta JMeter

Metric Name	IC_CM	IC_CC	IC_CC'	IC_CM + HITS	IC_CC + HITS	IC_CC' + HITS	Our Approach
Precision %	7	11	36	18	25	46	43

Graphical form of the above table is given in Figure 23.

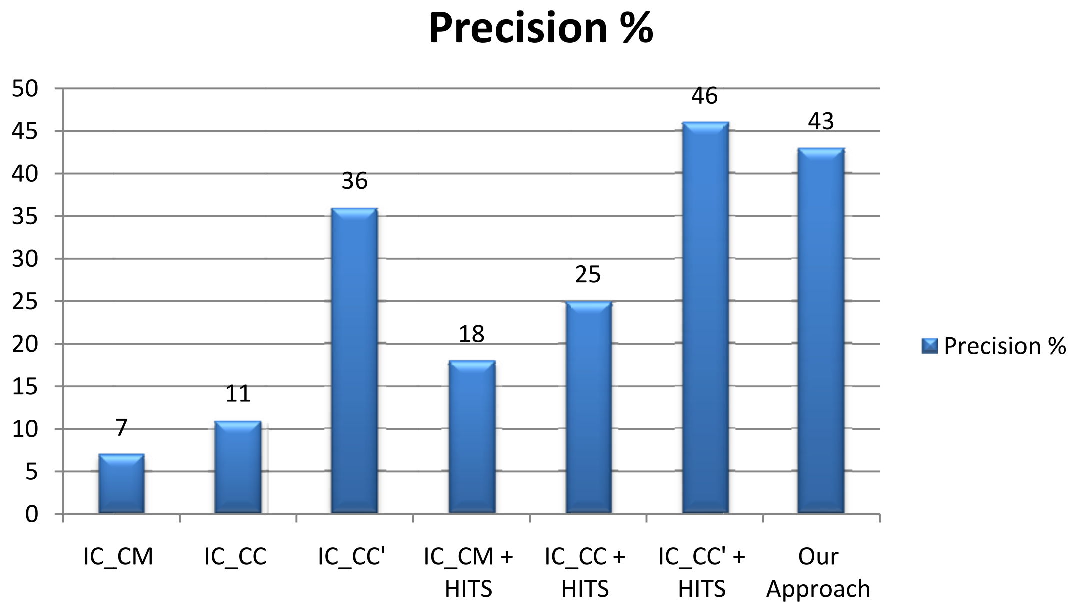


Figure 23: Comparison of Precision % -- Jakarta JMeter

6.4.2 Recall Comparison

Recall of our approach is compared with existing approaches in the following Table.

Table 22: Comparison of Recall % -- Jakarta JMeter

Metric Name	IC_CM	IC_CC	IC_CC'	IC_CM + HITS	IC_CC + HITS	IC_CC' + HITS	Our Approach
Recall %	14	21	71	36	50	93	86

Graphical form of the above table is given in Figure 24.

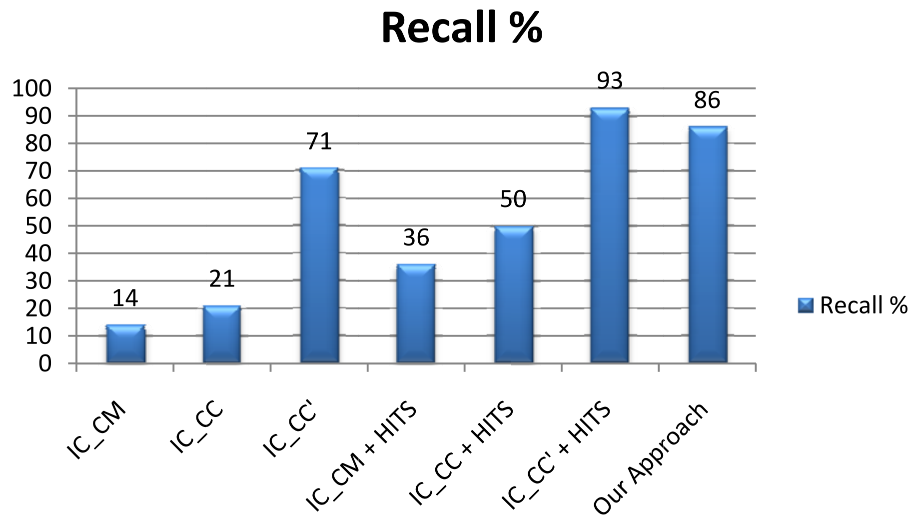


Figure 24: Comparison of Recall % -- Jakarta JMeter

6.4.3 Fallout Comparison

Fallout of our approach is compared with existing approaches in the following Table.

Table 23: Comparison of Fallout % -- Jakarta JMeter

Metric Name	IC_CM	IC_CC	IC_CC'	IC_CM + HITS	IC_CC + HITS	IC_CC' + HITS	Our Approach
Fallout %	93	89	64	82	75	54	57

Graphical form of the above table is given in Figure 25.

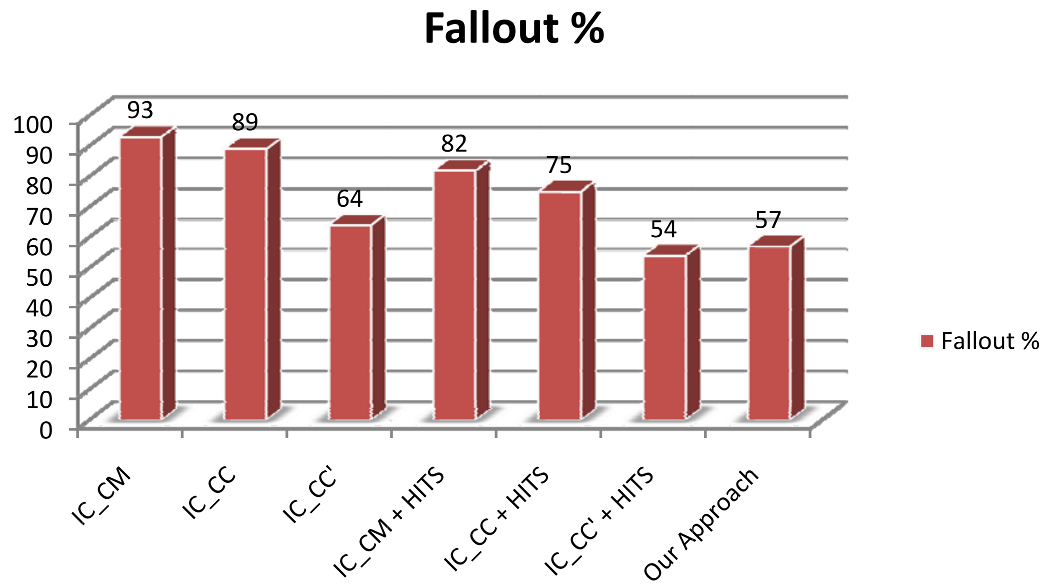


Figure 25: Comparison of Fallout % -- Jakarta JMeter

6.4.4 F-measure Comparison

The F-measure of our approach is compared with other approaches in the following Table.

Table 24: Comparison of F-Measure % -- Jakarta JMeter

Metric Name	IC_CM	IC_CC	IC_CC'	IC_CM + HITS	IC_CC + HITS	IC_CC' + HITS	Our Approach
F-Measure	25	45	45	37	52	61	58

Graphical form of the above table is given in Figure 26.

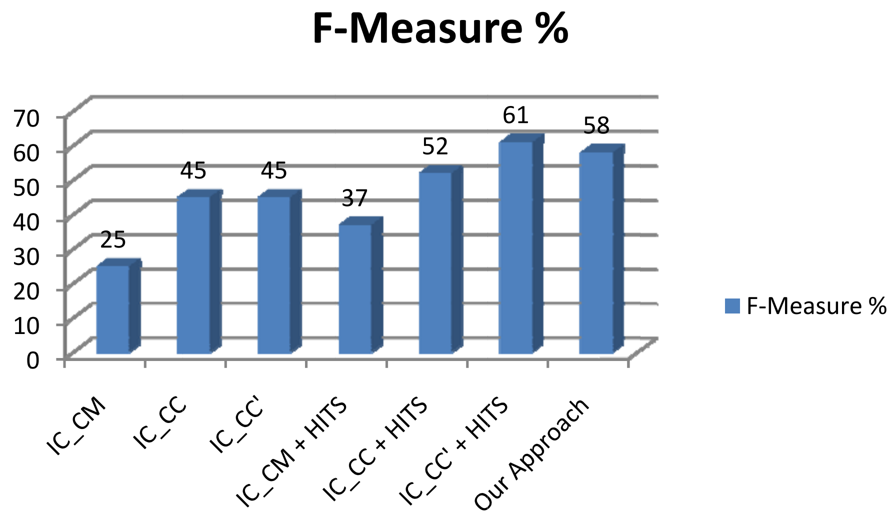


Figure 26: Comparison of F-Measure % -- Jakarta JMeter

6.4.5 Time Usage – (effort to perform the complete analysis, from start to finish)

In this criteria time is measured for the normal execution time taken by the machine to execute the use case without instrumentation as well as the time taken by the machine to execute the same use case with the overhead of instrumentation involved in our approach. The time for all existing approaches was taken from the published results of the experiments conducted by the respective author. Time measurement is taken as a relative measure, not as an absolute measure. For instance, the results of previous experiments reported that the original execution time for selected use case of Jakarta JMeter was 82 seconds; but with overhead involved in measuring the dynamic coupling information, the execution time elevated to 75 minutes (4500 seconds). It is evident that the existing approach has increased the original execution time by a factor of 54. With our approach the original execution time of same use case for Jakarta JMeter was 50 seconds; but with overhead involved in measuring the dynamic coupling information, the execution time elevated to 5 minutes (300 seconds). It is noticeable that our approach has increased the original execution time by a factor of 6 which is far more efficient than the factor of 54. On the basis of this principle all of the measurements are taken and their results shows that new technique presented in this thesis is far better than the existing one in term of time usage.

Table 25: Details of results for Time Usage – Jakarta JMeter

APPROACH	ORIGINAL EXECUTION TIME (SECONDS) *(A)*	EXECUTION TIME WITH APPROACH (SECONDS) *(B)*	OVERHEAD FACTOR *(B/A)*
IC_CM	82	4500	54
IC_CC	82	4500	54
IC_CC`	82	4500	54
IC_CM + HITS	82	4500	54
IC_CC + HITS	82	4500	54
IC_CC` + HITS	82	4500	54
Our Approach	50	300	6

Graphical form of the above table is given in Figure 27.

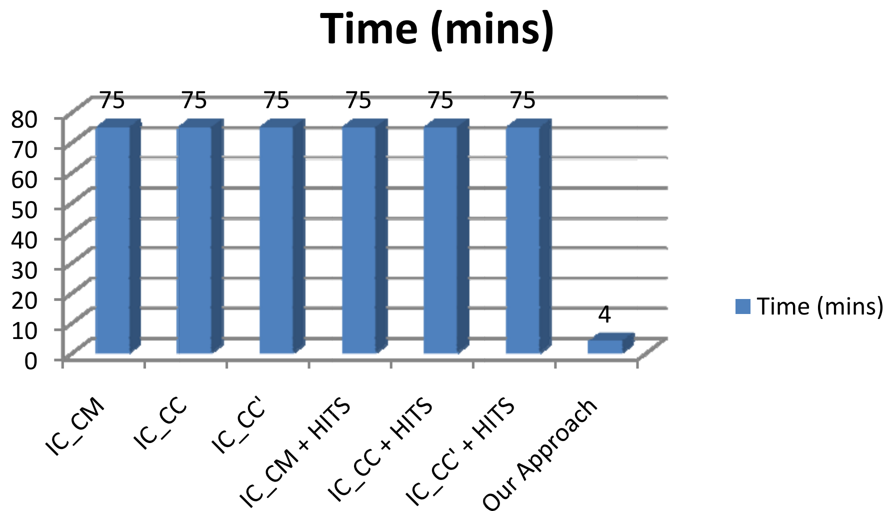


Figure 27: Comparison of time usage of approaches – Jakarta JMeter

6.5 An outline of achievements in this research

- A new variant of a dynamic coupling metric is introduced that takes into consideration the loading order of the class in the application.
- An implementation of the approach to calculate the new variant of the dynamic coupling metric during program execution is provided.
- The precision, recall and performance of our approach is compared with the other analogous experiments performed on the same software system.

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this chapter research work is concluded and some future directions are described. The chapter is of vital importance because it provides a bird's eye-view of the methodology and gives future directions for new researchers.

7.1 Conclusion

Traversing the entire set of classes in the project in order to find those few classes that provide substantial glimpse of the inner workings of the software is challenging for new programmers. Our technique for Discovering Key Classes can be of great help in this situation, since it provides the facility to identify automatically those few classes (from the entire set of classes in the system) that are ideal nominees for initial program comprehension. An open source software known as Apache Ant has been used for experimentation in our approach. This particular software was selected as case study because the results of other analogous experiments performed on the same system were available for comparison of our approach with other approaches. We have used the same baseline to evaluate our approach that was used by the authors in analogous experiments.

The concept of dynamic coupling and the analogy of core structure from civil engineering are fundamental to our proposed approach. On the basis of these two principles, our approach collects information from the guinea pig system (Apache Ant) at runtime. The results of our experiment have shown that using runtime coupling information and considering the loading order of the class in the application, we successfully recalled 90 percent of the key classes of the guinea pig system. In addition, the precision level of our approach was slightly under 50 percent with such level of recall.

In this research we have conceived and implemented a helpful technique for a software engineer who aims to get acquaintance with a completely new software

project. The key classes identified by our heuristic provide a miniature number of points to the user to pursue his quest for obtaining comprehensive understanding of the system.

7.2 Contribution

In this thesis, we have presented a new approach for coupling based analysis of object-oriented systems. The runtime coupling relationships among object-oriented software components and the loading order of the classes in the application is considered to calculate dynamic coupling metric. We have used dynamic analysis technique in our approach to instrument the object-oriented system. We have demonstrated an efficient approach to measure the runtime coupling relationships among object-oriented software components and successfully applied the measure to the problem of initial program comprehension.

We have also introduced a variant of dynamic coupling metric for object-oriented software. This coupling metric is derived from the framework of dynamic coupling by [10] with slight modification. An open source case study has been used to validate this metric and it has been demonstrated that this metric is useful in initial stages of program comprehension.

Another achievement is that we have practically applied the results of dynamic coupling metric to the initial program comprehension problem. During this research we have developed an algorithm to find efficient solution to the program comprehension problem utilizing dynamic coupling metric.

This research serves as a proof of concept that shows the pragmatism and efficacy of coupling-based investigation of software systems using an open source case study.

This thesis has produced the following publication:

- Muhammad Kamran, Farooque Azam, Assia Khanum, *“Discovering Core Architecture Classes to Assist Initial Program Comprehension”*, Lecture Notes in Electrical Engineering Volume 211, 2013, pp 3-10, Springer Berlin Heidelberg

7.3 Future Work

A number of research directions have already been identified by us for future work. These routes will allow the refinement in the validation of our approach for discovering key classes. In the first place, we want to conduct a controlled experiment that can be used to evaluate the significance of these key classes in the comprehension process of a large software system, when the programmer possesses no prior knowledge of the application. In addition, we plan to test the approach on a diverse set of applications belonging to different problem domains.

An attractive research direction for future could be the use of dynamic coupling for reverse engineering of applications. Sufficient details are being produced by this research through dynamic analysis to reverse engineer the design of software. To reverse engineer the structural characteristics of the system, the use of coupling information (including the component interactions) could be very helpful.

To assess the reusability of a class or the relationship of class with the relevant classes, ranking of classes is often helpful. A mechanism for ranking components of software is called *Component Rank* that analyzes real usage relationships among the components and promulgates their importance by means of usage relationships. Coupling measures can be utilized for ranking of components and/or classes. They should be able to identify the use of a component and any dependent components. Since the coupling metrics have the potential to detect how a class is utilized as well as its dependencies, they can be of great use for ranking classes with respect to the reusability. A tightly coupled component is less likely to be a reusable component, whereas a loosely coupled component can be an ideal candidate for reusable component.

REFERENCES

[1] Spinellis D. Code Reading: The Open Source Perspective. Addison-Wesley: Boston, MA, USA, 2003.

[2] Wilde N. Faster reuse and maintenance using software reconnaissance. Technical Report, Technical Report SERC-TR-75F, Software Engineering Research Center, CSE-301, University of Florida, CIS Department, Gainesville, FL 1994. URL citeseer.nj.nec.com/wilde94faster.html.

[3] Corbi TA. Program understanding: Challenge for the 90s. IBM Systems Journal 1990; 28(2):294–306.

[4] Biggerstaff TJ, Mitbander BG, Webster D. The concept assignment problem in program understanding. Proceedings of the International Conference on Software Engineering (ICSE), IEEE Computer Society: Los Alamitos, CA, USA, 1993; 482–498.

[5] Lakhotia A. Understanding someone else's code: Analysis of experiences. Journal of Systems and Software Dec 1993; 23(3):269–275.

[6] von Mayrhauser A, Vans AM. Program comprehension during software maintenance and evolution. IEEE Computer Aug 1995; 28(8):44–55.

[7] Robillard MP, Coelho W, Murphy GC. How effective developers investigate source code: an exploratory study. IEEE Transactions on Software Engineering 2004; 30(12):889–903.

[8] Tahvildari L, Kontogiannis K. Improving design quality using meta-pattern transformations: A metric-based approach. Journal of Software Maintenance and Evolution: Research and Practice 2004; 16(4–5):331–361.

[9] Chidamber SR, Kemerer CF. A metrics suite for object oriented design. IEEE Transactions on Software Engineering 1994; 20(6):476–493.

[10] Arisholm E, Briand L, Foyen A. Dynamic coupling measurement for object-oriented software. IEEE Transactions on Software Engineering 8 2004; 30(8):491–506.

[11] Zaidman A, Calders T, Demeyer S, Paredaens J. Applying webmining techniques to execution traces to support the program comprehension process.

Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR), IEEE Computer Society: Los Alamitos, CA, USA, 2005; 134–142.

[12] Zaidman, A. and Demeyer, S., Automatic identification of key classes in a software system using webmining techniques, Journal of Software Maintenance and Evolution: Research and Practice, vol. 20, no. 6, 2008, pp. 387-417.

[13] Briand LC, Daly JW, W¨ust JK. A unified framework for coupling measurement in object-oriented systems. IEEE Transactions on Software Engineering 1999; 25(1):91–121.

[14] Zaidman A, Du Bois B, Demeyer S. How webmining and coupling metrics can improve early program compehension. Proceedings of the International Conference on Program Comprehension (ICPC), IEEE Computer Society: Los Alamitos, CA, USA, 2006; 74–78.

[15] Robillard MP. Automatic generation of suggestions for program investigation. SIGSOFT Software Engineering Notes 2005; 30(5):11–20.

[16] Greevy O, Ducasse S. Correlating features and code using a compact two-sided trace analysis approach. Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR), IEEE Computer Society: Los Alamitos, CA, USA, 2005; 314–323.

[17] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Proceedings European Conference on Object-Oriented Programming, volume 1241 of LNCS, pages 220–242. Springer-Verlag.

[18] Storey, M.-A. D., Wong, K., and Muller, H. A. (2000). How do program understanding tools affect how programmers understand programs? Science of Computer Programming, 36(2–3):183–207.

[19] Renieris, M. and Reiss, S. P. (1999). ALMOST: Exploring program traces. In Proc. 1999 Workshop on New Paradigms in Information Visualization and Manipulation, pages 70–77. http://citeseer.nj.nec.com/renieris99almost.html.

[20] Pennington, N. (1987). Stimulus structures and mental pre-presentations in expert comprehension of computer programs. Cognitive Psychology, 19:295–341.
[21] Andrews, J. (1998). Testing using log file analysis: tools, methods, and issues. In Proceedings of the 13th International Conference on Automated Software Engineering (ASE'98), page 157. IEEE Computer Society.
[22] Yourdon, E. and Constantine, L. L. (1979). Structured Design: Fundamentals of a Discipline of Computer Program and System Design. Prentice Hall.
[23] M. Bauer. Analysing Software Systems by Using Combinations of Metrics[C]. In Proceedings of ECOOP'99 Workshops, Springer-Verlag LNCS 1743, Lisbon, 1999, 170-171.
[24] Andy Zaidman, Toon Calders, Serge Demeyer, and Jan Paredaens, Selective Introduction of Aspects for Program Comprehension, In Proceedings WARE'04 (WCRE Workshop on Aspect Reverse Engineering), 2004
[25] Kleinberg JM. Authoritative sources in a hyperlinked environment. Journal of t he AC M 1999; 46(5):604–632. URL citeseer.ist.psu.edu/article/ kleinberg98authoritative.html.
[26] Lehman M, Belady L. Program evolution: processes of software change. Academic Press Professional, Inc.:San Diego, CA, USA, 1985.
[27] Demeyer S, Ducasse S, Nierstrasz O. Object-Oriented Reengineering Patterns. Morgan Kaufmann, 2003.
[28] Ko AJ, Myers BA, Coblenz MJ, Aung HH. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Transactions on Software Engineering 2006; **32**:971–987.
[29] Systä T. On the relationships between static and dynamic models in reverse engineering java software. Proceedings of the Working Conference on Reverse Engineering (WCRE), IEEE Computer Society: Los Alamitos, CA, USA, 1999; 304–313.
[30] Eisenbarth T, Koschke R, Simon D. Locating features in source code. IEEE Transactions on Software Engineering 2003; **29**(3):210–224.

[31] Richner T, Ducasse S. Using dynamic information for the iterative recovery of collaborations and roles. Proceedings of the International Conference on Software Maintenance (ICSM), IEEE Computer Society: Los Alamitos, CA, USA, 2002; 34–43.

[32] Eisenbarth, T., Koschke, R., and Simon, D. (2001). Aiding program comprehension by static and dynamic feature analysis. In 17th International Conference on Software Maintenance (ICSM'01), pages 602–611. IEEE Computer Society.

[33] El-Ramly, M., Stroulia, E., and Sorenson, P. (2002). From run-time behavior to usage scenarios: an interaction-pattern mining approach. In Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining, pages 315–324. ACM Press.

[34] Gargiulo, J. and Mancoridis, S. (2001). Gadget: A tool for extracting the dynamic structure of java programs. In Proceedings of the Thirteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'01), pages 244–251.

[35] Gschwind, T., Oberleitner, J., and Pinzger, M. (2003). Using run-time data for program comprehension. In Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC'03), pages 245–250. IEEE Computer Society.

[36] Systä, T. (2000a). Static and Dynamic Reverse Engineering Techniques for Java Software Systems. PhD thesis, University of Tampere.

[37] P. Jalote. An Integrated Approach to Software Engineering. Springer-Verlag, New York NY, 1991.

[38] Jeff Offutt, Mary Jean Harrold, and P. Kolte. A software metric system for module coupling. The Journal of Systems and Software, 20(3):295-308, 1993.

[39] P. G. Frankl and E. J.Weyuker. An applicable family of data flow testing criteria. IEEE Transactions on Software Engineering, 14(10):1483-1498, 1988.

[40] Zhenyi Jin and Jeff Offutt. Coupling-based criteria for integration testing. The Journal of Software Testing, Verification, and Reliability, 8(3):133-154, 1998.

[41] Murphy GC, Notkin D, Sullivan K. Software reflexion models: bridging the gap between source and highlevel models. Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE), ACM: New York, NY, USA, 1995; 18–28.

[42] Walker RJ, Murphy GC, Freeman-Benson B, Wright D, Swanson D, Isaak J. Visualizing dynamic software system information through high-level models. Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM SIGPLAN Notices, vol. 33, ACM: New York, NY, USA, 1998; 271–238.

[43] Ducasse S, Lanza M, Bertuli R. High-level polymetric views of condensed run-time information. Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR), IEEE Computer Society: Los Alamitos, CA, USA, 2004; 309–318.

[44] Greevy O, Lanza M, Wysseier C. Visualizing live software systems in 3D. Proceedings of the Symposium on Software visualization (SoftVis), ACM: New York, NY, USA, 2006; 47–56.

[45] Muhammad Kamran, Farooque Azam, Asia Khanum, "Discovering Core Architecture Classes to Assist Initial Program Comprehension", Lecture Notes in Electrical Engineering Volume 211, 2013, pp 3-10, Springer Berlin Heidelberg.