Supporting Source Code Feature Analysis Using Execution Trace Mining

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Mohammad Asif A. Khan

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

Software maintenance is a significant phase of a software life-cycle. Once a system is developed the main focus shifts to maintenance to keep the system up to date. A system may be changed for various reasons such as fulfilling customer requirements, fixing bugs or optimizing existing code. Code needs to be studied and understood before any modification is done to it. Understanding code is a time intensive and often complicated part of software maintenance that is supported by documentation and various tools such as profilers, debuggers and source code analysis techniques. However, most of the tools fail to assist in locating the portions of the code that implement the functionality the software developer is focusing. Mining execution traces can help developers identify parts of the source code specific to the functionality of interest and at the same time help them understand the behaviour of the code.

We propose a use-driven hybrid framework of static and dynamic analyses to mine and manage execution traces to support software developers in understanding how the system's functionality is implemented through feature analysis. We express a system's use as a set of tests. In our approach, we develop a set of uses that represents how a system is used or how a user uses some specific functionality. Each use set describes a user's interaction with the system. To manage large and complex traces we organize them by system use and segment them by user interface events. The segmented traces are also clustered based on internal and external method types. The clusters are further categorized into groups based on application programming interfaces and active clones. To further support comprehension we propose a taxonomy of metrics which are used to quantify the trace.

To validate the framework we built a tool called TrAM that implements trace mining and provides visualization features. It can quantify the trace method information, mine similar code fragments called active clones, cluster methods based on types, categorise them based on groups and quantify their behavioural aspects using a set of metrics. The tool also lets the users visualize the design and implementation of a system using images, filtering, grouping, event and system use, and present them with values calculated using trace, group, clone and method metrics. We also conducted a case study on five different subject systems using the tool to determine the dynamic properties of the source code clones at runtime and answer three research questions using our findings. We compared our tool with trace mining tools and profilers in terms of features, and scenarios. Finally, we evaluated TrAM by conducting a user study on its effectiveness, usability and information management.

Acknowledgements

Firstly, I would like to express my profound gratefulness and honor to my supervisors Dr. Kevin Schneider and Dr. Chanchal Roy, for their continuous support, advice and care throughout my entire duration of study. Their endless patience, scholarly guidance, continual encouragement, constant and energetic supervision, constructive criticism, valuable advice, at all stages have made it possible to complete this thesis. I would like to express my gratitude to all the members of my advisory committee: Dr. Ralph Deters, Dr. Gord McCalla and Dr. Nurul A. Chowdhury for their valuable suggestions.

Also, I would like to appreciate the assistance of Ms. Jan Thompson and Ms. Gwen Lancaster, at the Department of Computer Science, University of Saskatchewan, for their support and generosity.

Thanks to all students of the Software Research Lab (SRLab) for their friendship and support. I would also like to extend my gratitude to the members of the MADMUC lab for their continual support. I am grateful to the members of the DISCUS lab for extending their hands too.

Finally, I would like to express my deepest affection and gratefulness to my beloved wife, Shomoyita Jamal without whom the journey would not have been possible, my parents, and in-laws for the unconditional love and support throughout this period.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Software engineering is a multidisciplinary area comprising phases of design, development, testing and maintenance of a system. Once a system is developed, the main focus and challenge lie in the maintenance phase. The objective of this phase is to support the continuous change requirements during the evolution of software. Such changes are inevitable and can be triggered in order to fix bugs or optimize code. Customers often change their requirements which may require changes both at the design and implementation levels, but may also propagate to the maintenance phase. It has been found that 50% to 75% [106] of software cost is dedicated to the maintenance phase.

Before making any change in a software system, it is required to comprehend it well enough. This ensures successful integration of the required changes. However, studies show that program comprehension is an intensive and time consuming process, occupying 50-60% [7, 10, 15] of software engineering efforts. Developers [49] must go through the time consuming task of knowing and gathering a variety of underlying information about the system before changing it. Since the demand of maintenance, reuse and re-engineering has grown over time, program comprehension has become a challenge for engineers. Comprehension includes identifying a system's building blocks, their interrelationships and even their behaviour in terms of context of use. We define context as the objectives of an implementation of a specific portion of code. In this thesis, we consider methods as the building blocks of an implementation of a software feature and therefore, we determine their context by analysing features in order to support program comprehension. A method can be defined as a named block associated with a type, with or without parameters and enclosing a set of statements, expressions or variables. Every method is responsible for performing a certain functionality.

Maintenance engineers often resort to analysing documentation, studying the source code or running the system for revealing the underlying structure and functionality. For instance, consider a scenario when a developer is given the task of adding a new feature to a system developed by someone as requested by a client. The options he may look for, to understand the system is to ask the developers to explain the design and implementation of the system, to look at the documentation or to utilise other source code analysis techniques. In number of cases the developer will be unavailable to ask. In the case of legacy systems, the process becomes even more difficult due to the lack of proper and updated documentation caused during

the evolution of a system over time and having passed through many developers. This makes maintenance more problematic and difficult. In other words, we can say that the system needs to be reverse engineered or re-documented. Although comprehension can be aided by the use of profilers or debuggers, the process itself becomes too time-consuming and painstaking where a lot of different variables and program states need to be taken care of at the same time.

One way to support program comprehension is to understand what features are implemented, how they are implemented and where they are located which we together term as feature identification. Every feature when aggregated and connected together results in a system that is usable for the user. Evolution of a system due to routine maintenance activities results in certain impacts on the features. Every feature is a collection of methods that are tied together during the implementation. Portions of code executing during runtime depend solely on the choice of uses made by the users corresponding to specific features in the systems. These features when used result in the execution of methods that implemented them. Therefore, identifying underlying portions of code that implements specific features and gathering their behavioural information through mining can assist with comprehension. Locating and understanding how the method blocks specific to features have been implemented can contribute to the maintenance task such as the addition of new code (statements, function call and so on),the deletion of code or the optimization of the existing block or method with new code. Now, locating only the method blocks cannot provide insights into why the specific portions of code have been implemented. Therefore, directing the developers in knowing the objective of the implemented code can assist them analyse and understand the code further.

Linking code associated with specific features can be achieved through dynamic analysis. Runtime data, called execution traces, containing sequences of method calls collected during the execution of a system when mined can help in providing deep insights into the implementation level. Execution traces, if generated based on system use can make the comprehension clearer and even modular with every module containing a set of method calls representing a specific feature. Such modular or segmented traces can be collected using the notion of software testing [31]. By definition testing is a form of dynamic analysis that requires execution of a system with a set of inputs called tests and examining the result to get a specification of the actual program behaviour.

Software systems often contain similar code fragments, also known as code clones [74]. Existing clone detection tools can identify code fragments that are textually similar but fail to provide the context in which they are used. Knowing the portion of code specific to features can help us in understanding how syntactically similar code fragments behave and in what contexts they have been used. For instance, a clone class can contain a number of similar fragments but they may be used in different contexts. The code fragments may be syntactically the same but may provide different functionality, such as drawing a rectangle and drawing a square. This opens the avenue for mining execution traces in order to determine the context of use of duplicated code. On the other hand, clone management is an active software engineering research area that has contributed a diverse set of tools and algorithms to aid in the management and maintenance of software.

Code duplication is a common programming practice. Programmers use code duplication to increase the speed of the software development process by using similar codes to implement common functionalities. However, if the copied code is not tested for bugs it will lead to their propagation and eventually render the system more defective and vulnerable. Fixing a bug in one clone fragment may require reviewing and perhaps applying a similar fix to its other clone fragments. Any inconsistency in making a change in a set of clone fragments may introduce new bugs. Consequently, clone management being a subset of software maintenance will become too complicated in the course of time. Mining runtime information of duplicated code fragments using a set of metrics and analysing their behaviour can further assist in prioritizing inspection of clones during the maintenance phase.

1.2 Problem Definition

Analysing execution traces to support software maintenance [62, 37, 38] and trace visualization tools [91, 44, 48, 94] have been studied before by many authors. A number of techniques and tools have been developed as well. However, due to the complex nature of the execution traces that are obtained, the analysis is challenging and difficult. As a result, there is a demanding need for more meaningful and informative analysis techniques that can help in program comprehension and the maintenance task in hand. A number of trace mining metrics [26], a compression technique [27] and a trace partitioning technique [89, 66, 67, 68] have already been proposed. From the literature we have identified that:

1 Once a system is reverse engineered using dynamic analysis, the execution pattern is recorded in the form of execution traces containing method calls. Although traces provide us with a sequence of method calls, understanding the context in which they are used can make comprehension easier and simpler. Traces when mined further using metrics can enhance comprehension by providing behavioural aspects of the features executed by a user.

2 Although a number of trace mining metrics [26] have been proposed, no study on assessing the dynamic behaviour of similar code fragments, called clones has been attempted. Therefore, to what extent clones are active and how they can be used in maintenance has not been studied before. It is indeed a necessary and an important aspect to highlight the presence of clones and mine their information with new sets of metrics. Clones do have impact on maintenance activity and although there are a lot of detection techniques, there is little work in discovering their run time effect on the system with respect to the context in which they are used.

3 Execution traces are a highly useful and informative way of analysing a system to support comprehension and debugging. However, the main challenge when dealing with such dynamic information is its sheer volume and diversity. A small system with only a few hundred lines of code when instrumented provides a considerable amount of traces (method calls). Trace management thus plays a vital

role in such a situation. Trace measurement techniques in the form of metrics [26] followed by trace comprehension techniques have been proposed but there is still scope for further mining to make them more understandable. Although trace partitioning using user interface events has also been proposed in [89], tool support with trace exploration facilities in a more precise and modular way with features incorporating interface screen shots, filtering, categorization in a simple and interactive way would add benefit to users during feature analysis.

1.3 Contribution

The main intent of this thesis is to support program comprehension by using traces to locate features, their corresponding source code and mining the traces to extract feature based dynamic information of the system. We adapt a use-driven hybrid approach of static and dynamic analyses to mine a system's runtime data and traces to determine the implementation detail of a system. We use a use-driven approach and express a system's use as a set of tests. In our approach we develop a set of uses that represents how a system is used or how the user uses some specific functionalities. Each use set describes a user's interaction with the system. To manage large and complex traces, they are modularized through segmentations based on system use, tests and user interface events. The segmented traces are clustered based on internal and external methods. The clusters are further categorized into external groups called application programming interface and active clones. To further support the comprehension process the trace components are quantified using a set of active metrics such as trace, method, clone and group metrics which we have proposed.

To validate the framework we built a tool called TrAM that implements the trace mining and the visualization aspects. It can quantify the trace method information, mine similar code fragments called active clones, cluster methods based on types, categorise them based on groups and quantify their behavioural aspects using a set of metrics. The tool also lets the users visualize the design and implementation of a system using images, filtering, grouping, event and system use selection, and present them with values calculated using trace, group, clone and method metrics. The tool would assist the developers/maintainers in the following ways:

a. Allow the developers to understand the implementation details of a system's functionalities by running the system based on system use, tests and events. Each time the developers try to study the system, they can create their own system use and collect the run time data accordingly. The trace segmentation based on system use, tests and events supported through the process of annotation will help them to understand the context of use of the implemented code and guide them towards code modifications.

b. Allow the developers to understand the contexts in which clone fragments have been used and their behaviour in terms of metrics. For instance, a clone class contains clone fragments and the developer wish to find out in how many tests a particular clone fragments have executed or how active are the clone classes. All the fragments belonging to a class may execute or only a few of them may be active.

c. Allow the developers to find the consumption of APIs and figure out the most frequently used ones based on contexts. For instance, a developer might want to know which type of API (String, IO, or Util) have been used most and in which contexts.

d. Allow the developers to map high level view, for example user interface, with the source code easily using snapshot view, code view and trace exploration technique. For instance, a developer wish to know about the start and the end of a particular feature execution with the sequences of method calls contained in it. The developer may also want to know only the method calls associated with a particular button click that performs a certain function. This will allow them to understand how a particular feature corresponding to that specific button has been implemented. Therefore, if any change is required to be made, only these method calls associated with the button needs to be focused on.

e. Allow the developers to analyse the behaviour of a system using a set of metrics based on system use, events, methods and groups. For instance, a developer may wish to know the total number of files invoked, total time consumed by the method calls and so on for a particular test or event such as a button click event during a draw triangle test. A developer may want to know about the total time consumed in the execution of a certain functionality which can be determined through the segmentation of trace based on tests and events.

We also conducted a case study on five different subject systems using the framework to determine the dynamic properties of the source code clones at runtime and answer three research questions: a) To what extent are clones active during run time? b) How active are the active clones and c) Does the active clone identification support software maintenance activities?. The results of our study did answer the research questions. The findings indicated that for each system we were able to find a considerable amount of clones as active and occupying a major portion of CPU time. The active clone genealogy, when considered over versions for each system also indicated that the active clones change more inconsistently. A detailed insight into the active clone genealogy revealed that clone inspection during clone management could be prioritized based on clone types such that Type 3 clones should be managed first followed by Type 2 and Type 1 clones. We also compared our tool with other existing trace mining tools and profilers in terms of features and scenarios. We found that our tool is stronger for locating features in terms of trace segmentation, snapshot view, code view, calculating metrics and categorization. Finally, a brief user study was accompanied by the evaluation of TrAM on its effectiveness, usability and information management.

The rest of the thesis is organized as follows: Chapter 2 provides a brief overview of the background and the related studies. Chapter 3 describes the framework of our study, taxonomy of metrics, frameworks the schema model of our tool TrAM. Chapter 4 describes the implementation of TrAM followed by its evaluation using a comparison of features and scenarios with other existing trace analysis tools, profilers and user feedbacks. Chapter 5 presents the the empirical study on the runtime source code clone and answers to the research questions. Finally the thesis is concluded by a discussion and indication of future work.

Chapter 2

Background and Related Work

This chapter presents the background and research relevant to the thesis. Since this thesis proposes a hybrid automated framework for collecting and analysing execution traces in terms of the source code, APIs and clones; the concept of software testing, execution traces, APIs and clones are discussed.

2.1 Software Maintenance

Software maintenance is a continuous process that commences right after software is delivered to the customers and is subdivided into four categories as follows:

a. Corrective Maintenance involves fixing the system caused due to bugs or faults in it.

b. Preventive Maintenance involves prevention of system failure beforehand to ensure that no faults could cause the system to fail.

c. Adaptive Maintenance involves bringing modifications in a system to meet the new requirements of the customers.

d. Perfective Maintenance involves in the improvements of the existing system so that further modifications such as addition of new features can be easily done.

2.2 Program Comprehension

Software maintenance, being a major process as mentioned in the previous section, requires vivid understanding about the system. The more a system could easily and simply be understood, the more effectively it can be maintained and utilised. Thus, a program can be understood based on the four strategies [100] mentioned in the following subsection:

2.2.1 Bottom-up Model

Bottom-up model is a technique in which a software engineer builds an abstract model of the source code for its comprehension. This process involves reading the source code and mentally grouping together low-

level programming details, called chunks, in the form of higher-level domain concepts. The whole procedure continues until an adequate understanding of the program is achieved.

2.2.2 Top-down Model

The main idea behind the top-down comprehension process is that it is totally dependent on hypothesis. During this approach, the software engineer first creates a hypothesis about the functionality of the system and then verifies the validity of it. Eventually a hierarchy of hypothesis is created until the low level hierarchy is matched with the source code. Such type of comprehension technique is usually performed by engineers who have a good knowledge about the functionality of the system.

2.2.3 Integrated Model

The integrated model is a hybrid one combining both the bottom-up and the top-down approaches. A number of studies were conducted on maintenance activities including adaptive, corrective and re-engineering. The results revealed that the software engineers have the tendency to switch among the comprehension strategies based on their expertise on the system and the code under investigation.

2.3 Reverse Engineering

Reverse engineering a process in which the subject system under investigation is analysed to determine its components and discover the inter-relationships among the components to create a representation of the system in another form of higher level of abstraction. It does not alter the structure of the subject system but facilitates the process of examination of the system. The three basic activities of the reverse engineering technique are:

. **Data gathering:** It is the initial step which involves collecting data in one of the following ways such as source code analysis using parsing, dynamic analysis using profilers and informal extraction of data using interview. The data obtained assists in identifying the system components and relationships among them.

. **Knowledge organization:** The raw data that are obtained must be represented in such a way so that program comprehension is achieved by allowing analysis of the components with the relationships among them and reflecting the users' perspective about the characteristic of the system. The representation is achieved using a data model that captures the static and dynamic information of an application required to support the data-related activities.

. **Information exploration:** This phase is the key contributor of the program comprehension. Once the data is collected and organized using a data model there must be a way for navigation so that the

subject system can be analysed. The navigation is based on domain and presentation specific criteria which signifies the information.

2.4 Software Testing

Software testing is an important process done to ensure the quality of software. The main intent of testing is to locate errors or faults during the software development life cycle. The testing activities [31] are composed of designing test cases, executing the software with the test cases and examining the result obtained through the execution of the software. The output thus obtained provides a specification of the actual program behaviour. Figure 2.1 shows the process of software testing that is adopted during the software maintenance. The model consists of a configuration of test inputs including test case design and creation, execution of software with the test inputs and finally evaluation of software based on the output of the execution. In the initial step, test configuration and software configuration are the two key aspects of the initial testing phase. Test configuration includes designing of test case, which technique to adopt and what tools to use. However, selection of test case depends on which type of techniques are used for testing. Software configuration consists of requirement specification, design specification, source code and so on. In the second phase, the output of the tests are evaluated based on the expected results which are termed as test oracles. It includes simulated result, hand calculated values and design specifications. After the evaluation phase, the final phase includes debugging and reliability prediction. During the debugging phase, the errors identified at the evaluation phase are taken into consideration and necessary corrections are done. At the same time the error rate data is used to predict the reliability of the software.

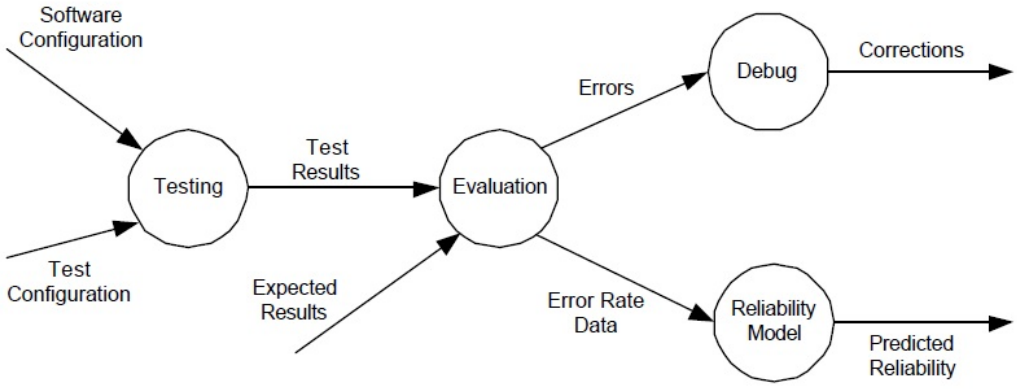

Figure 2.1: Testing Information Flow (taken from [53])

2.5 Dynamic Analysis

Dynamic analysis [85, 36, 5] is a technique of understanding the behaviour of a system by putting the system under execution. It is the only analysis technique [14] that reveals the actual behaviour of the system down from class level to architectural level. The main limitation of dynamic analysis is that it cannot prove that a program satisfies a particular property, however it can identify discrepancies in the properties and provide insights to the programmers about the behaviour of their programs. This technique requires three steps as follows: a)what inputs to provide, b) how to execute the system and c) what to look for in the analysis? It is an active analysis method that are often used during reverse engineering and program comprehension.

Dynamic analysis has its advantages and disadvantages [19]. The main advantages are: a) it is precise because unlike static analysis it can work without abstraction or approximation such as a particular sequence of statements in a path may execute or not and thus dynamic analysis provides a clear picture of it. Above all, it does provide the actual scenario of the control paths executed, values computed, memory consumed and so on. On the other hand, dynamic analysis suffers from incomplete program executions [20, 5]. A program may contain a number of execution paths connected by conditional statements (switch, if-then-else). As a consequence, all the branches will not logically execute at a time because they are mutually exclusive. In such cases execution of each branch will require different input sets. So this does not guarantee that a program executed using a particular set of inputs is representative of all the possible executions of the program. The prime challenge in dynamic analysis is choosing a good set of test suites with wide range of coverage that will exercise all the possible branches present in the program structure. However, through this one can easily relate the inputs used, the behavioural changes of the internal program and its output since it is straight forward and directly linked with the program execution.

2.6 Runtime Data Collection

One of the prime challenges in dynamic analysis is to collect runtime informations. The strategies followed for collections of data are: a) code instrumentation and b) profiling and debugging.

One of the techniques for collecting run time data is by instrumenting the source code. In source code instrumentation, probes are inserted into the source code. For example print statements may be inserted in various locations so that when the system runs the output of the print statements are logged in a file. In the case of an object-oriented system probes can be inserted at the entry and exit of every method to identify the start and the end of the method execution. The instrumentation process usually is done automatically. The second technique used to collect run time data is by instrumenting the system's execution environment such as Java Virtual Machine to generate logs based on user's interest [29]. For instance, JVMPI (Java Virtual Machine Profiler Interface)[50] is a binary function call interface between JVM and a profiler agent. A profiler agent, on the other hand connects JVM and the profiler front-end. Profilers such as hprof make use of the

JVM environment to generate events, stack traces, CPU usage, call graph and so on. The main advantage of this technique is that source code can be kept as it is with no further modifications. However, it provides only a fixed set of interfaces and often reduces performance. DTrace [107], a dynamic tracing framework allows logging of various behaviour of a system such as memory usage, CPU time, filesystems and so on. It defines various types of probes (module, function, name, provider) that can be used at the command line to instrument a system. Besides, debuggers [29] can also assist in the collection of run time data for analysis. The program to be tested must be executed under the control of the debugger that is available through any software development kit such as Eclipse, Netbeans and so on. To collect data breakpoints needs to be set at desired locations such as entry and exit of methods. This technique has its own advantages where both the source code and the runtime environment requires no modifications. However, the only disadvantage is that the execution of the system can be slowed down considerably.

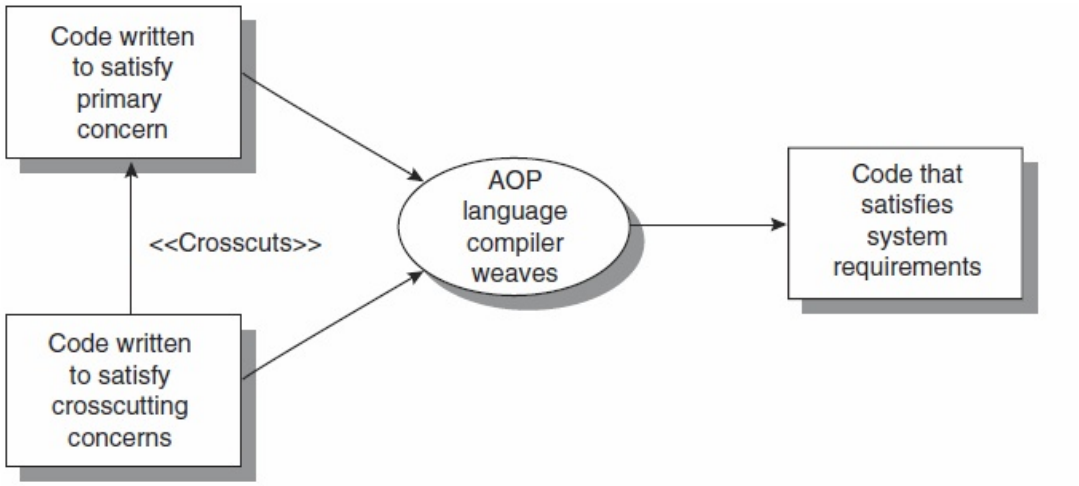

Figure 2.2: AOP Compilation (taken from [24])

2.6.1 Aspect Oriented Programming

Aspect Oriented programming [42, 34] is a technique used to separate the cross-cutting concerns in software systems. A concern is a requirement or functionality implemented in the code to achieve a particular goal. A system may have numerous concerns implemented in it for the system to meet the user requirements. All the programming implementations have the notion of modularizing the code or design into units or functions which collectively help to produce a complete software. Such modules are often termed as components and each concern of a system is encapsulated into components. As a result they allow the developers to achieve modularity and thus reduce complexity during maintenance. In case of object-oriented programming, the modularity is achieved through encapsulating data and the concerns into a conceptual unit called object. Although object-oriented programming has advantages such as inheritance, polymorphism and code reuse it has its own disadvantages.

The main disadvantage of OOP [34] is that although it provides a hierarchical structure for modularity some concerns are unable to be modularized in complex systems. Examples of which include logging, error handling, or synchronization and their implementations are spread over multiple modules i.e., number of methods and number of classes. Such types of concerns are said to be cross-cutting across the systems and the concerns are termed as cross-cutting concerns. AOP thus provides a means to program the cross-cutting concerns in a modular way which in turn helps to achieve the benefits of modularity. AOP provides modularity of code by removing code tangling and code scattering [24]. Aspect mining is a technique that is used to identify the cross-cutting concerns in a system. A number of techniques exists for mining the cross-cutting concerns and to evaluate their strengths and weaknesses an empirical study on three different subject systems has been conducted by Roy et al. [80].

Code tangling is a phenomenon when too many concerns are implemented into one module while code scattering is the scenario where one concern is implemented over various modules. Such implementation further induces situations like a) poor traceability - causing poor mapping between the concern and the implementation, b) decreased productivity - a developer has difficulty concentrating on one concern, c) provides less code reuse - a single module implements more than one concern tangled with each other such that a single concern of interest cannot be used, d) poor code quality - where debugging becomes difficult and the implementation carries bugs that may not be identified, and e) difficult evolution - situation when changes in one concern may produce side effects on other modules. Thus, AOP tends to solve these problems by separating the cross-cutting concerns into separate modules called aspects.

Implementation

One implementation of AOP in Java is achieved by coding concerns using a language called AspectJ [42]. Figure 2.2 shows the AOP implementation process. The AspectJ program contains base code and aspect code. The base code contains classes, interfaces, and other constructs that implements a given concern required by a system while the aspect code comprises of constructs needed to implement the cross-cutting concerns. The AOP compiler often known as weaver takes the base program and the aspect program as inputs and compiles them to weave the aspect concerns into the base program. The weaving [46] can be achieved in three different ways: a) source weaving, b) binary weaving, and c) load-time weaving.

Source weaving which is done during compile time takes the source code of the aspect and the base program to produce a byte code which is compliant with the Java byte code. In such case all the source code must be present together. In binary weaving the inputs to the weaver are in the byte code form. The byte codes are compiled separately using a Java or AspectJ compiler. For instance jar files or the class files can be used as an input to the weaver. However, unlike the source and binary weaving, load time weaving uses classes, aspects and configuration files defined in XML format. The load time agent can be of any of the form such as Java Virtual Machine Tool Interface (JVMTI) [46] agent, a classloader, or a Virtual Machine and application server-specific preprocessor.

Language Constructs

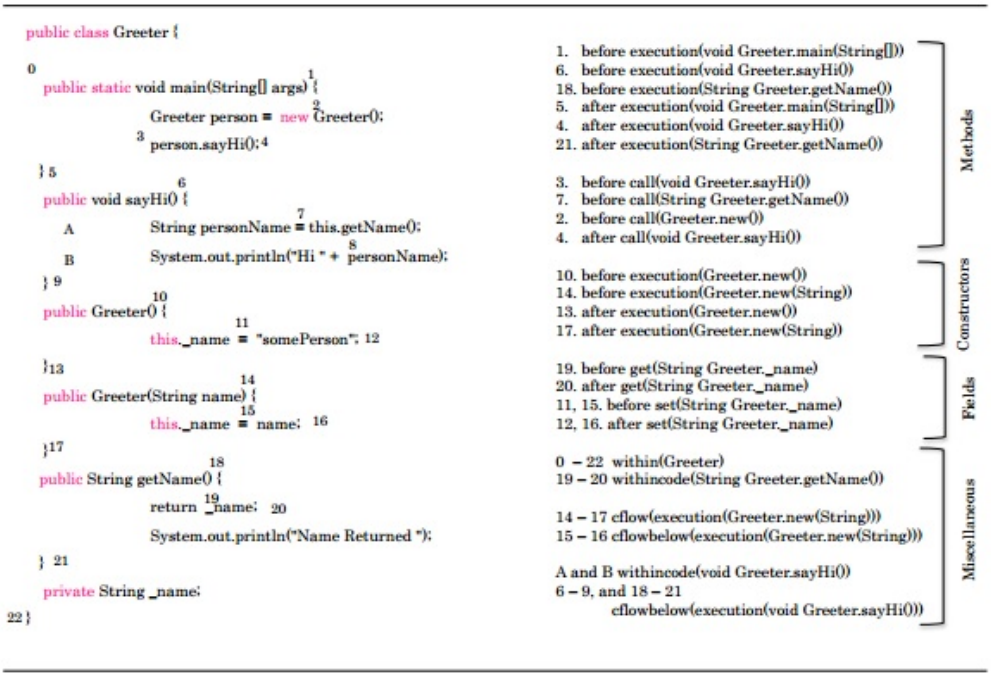

Figure 2.3: Example of Aspect Constructs (taken from [61])

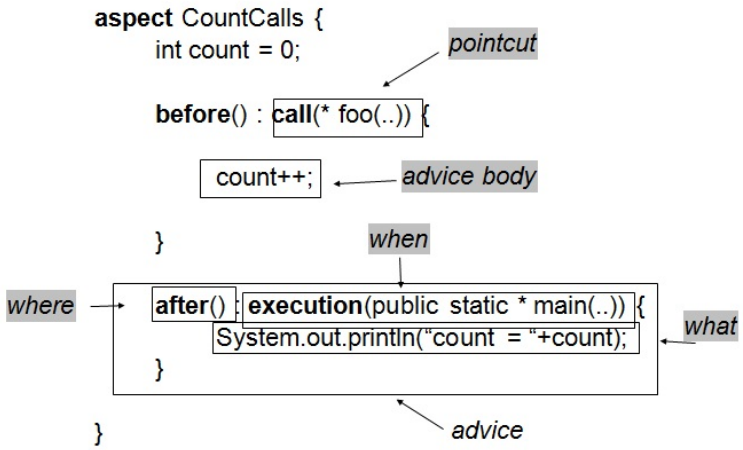

Figure 2.4: Implementation of AspectJ (taken from[1])

The AspectJ language [42] as shown in Figure 2.4[1] is mainly based on the concept of the joint point model. A joint point is the reference which allows the definition of structure of the cross-cutting concern in aspect modules. It is a well defined points in program where the aspects and the components join. The AOP further consists of four constructs: a) pointcut, b) advice, c) aspects, and d) inter-type declaration.

[1] cs.uwindsor.ca/ jlu/440/440AOP1-2011.pptx

Join point category	Join point types
Execution	Method execution Initializer execution Constructor execution Static initializer execution Handler execution Object initialization
Call	Method call Constructor call Object pre-initialization
Field access	Field reference Field assignment

Figure 2.5: Summary of Join Point Category (taken from [47])

Pointcuts are constructs that identify the subsets of join points defined in the program. The AspectJ language [24] supports a number of different types of join points such as method call, constructor call, and field access call as shown in Figure 2.5. Once the join points are identified, pointcuts are then used to select them using the designator $x(joinpointsignature)$ where x represents type of pointcuts containing the join point within it. For instance a method execution join point can be stated as $execution(signature)$ where the signature can be defined as $(type\_of\_methodclass.method\_name(parameterlist))$. Wild cards such as (*) and (..) can also be used while defining the designators to give them a type of property based on the pointcut. The (*) indicates all and (..) indicates any existing parameters in the method regardless of return types, declaring types, method names, and method parameters. A number of pointcuts can be combined together using the AND, OR and NOT operators to form an anonymous and a named user-defined composite pointcuts.

An advice defines the code to execute upon reaching the join points selected by the associated pointcut. It can be classified into three types such as before, after and around. Before() and after() advices are responsible for such that at which point during the join point computation they should execute. For instance, before() executes as the join point is reached. It has no additional matching criteria nor does it have any capability to alter the join point context. On the other hand, after advice executes after the join point is reached and all the processing within the method is completed. Again after() advice can further be classified into three types: unqualified which runs no matter what the outcome of the join point is, after throwing executes if the join point ends by throwing exception, and after returning executes only when the join point execution is successful and returns normally with no errors. Unlike before and after advice the around advice is a pre-emptible one which has the capability to stop at the join point and continue execution just after the join point. In summary, an advice answers the question of what to execute, when to execute and where to execute. Figure 2.3 illustrates the structure of an aspect, join point, pointcut and an advice in an AspectJ implementation while Figure 2.5 illustrates different kinds of join points in the program that are targeted by the aspects.

Aspects are the modular units of cross-cutting concerns that consists of join point, pointcuts and advice. An inter-type declaration is a static cross-cutting concern that can modify the static structure of classes, interfaces and aspects in the system.

2.7 Trace Analysis Tools

This section provides an overview of dynamic analysis tools and profilers that assist in trace collection, trace management and visualisation.

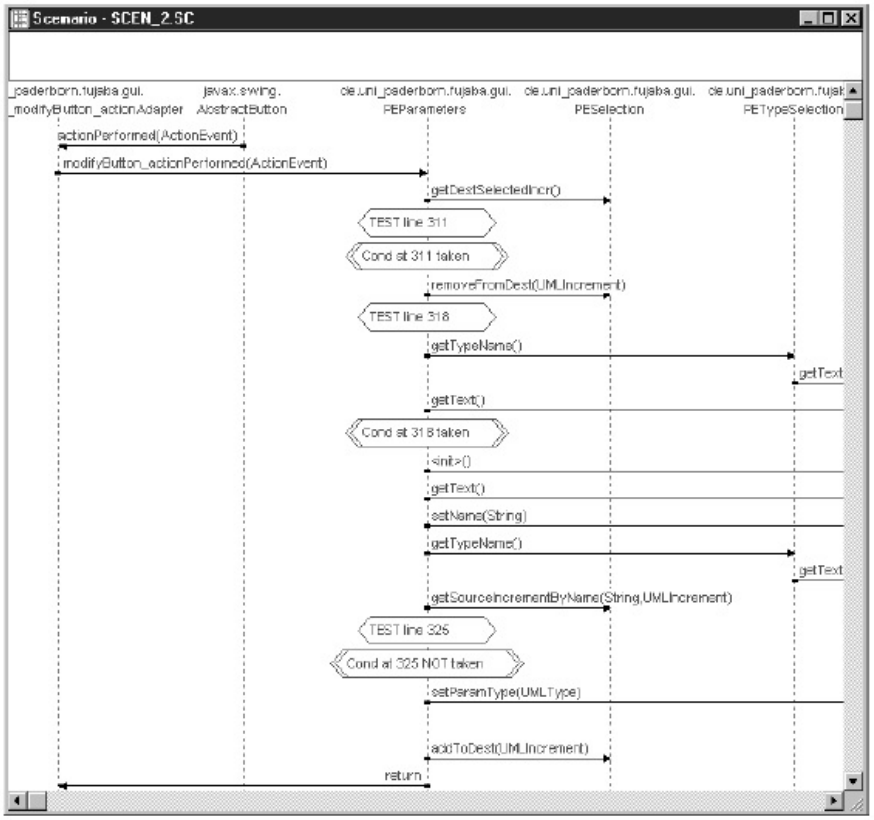

Figure 2.6: Diagram of Shimba (taken from [92]) Showing Execution Path of a Method Using a Sequence Diagram

2.7.1 Shimba

Shimba presents a reverse engineering environment [92] to comprehend Java system behaviours using a hybrid approach of static and dynamic analysis. The benefit of static analysis is that it permits the selection of components that will be used further using the analysis dynamically. It is assumed that tracing of the whole system is not required, if the engineers only tend to analyse a specific portion of it. The principal functioning of the environment is that it extracts system components such as class, interface, method, constructor, variable and static initialization blocks. It also picks interdependencies among the components, inheritance

relationships, containment relationships (e.g., a class contains a method), call relationships and so on. These information are then fed into another reverse engineering tool called Rigi where the components and the interdependencies are visualised using a directed graph containing nodes and directed edges respectively. Rigi also allows the users to be selective in node investigations by including only those that are of interest to them by running scripts.

So far, we have the environment to do the analysis, the component extractor and therefore, there is a need for a trace analyser. This is accomplished by the forward engineering tool SCED [45]. It provides a basis for constructing scenario diagrams, state diagrams and synthesis of state machines from them. Figure 2.6 provides an overview of the representation of the execution path of a method using a sequence diagram. The repeated sequence of identical events called behaviour patterns are also adapted by SCED using a string matching algorithm. Two types of patterns are identified: contiguous sequence due to loops shown as repetition constructs and the non-contiguous ones shown using sub-scenario constructs with boxes containing the events associated with the patterns. Out tool, TrAM adapts both the static and dynamic analysis to extract static and dynamic information separately. Using static analysis we mine the source code to extract the methods and detect similar code fragments called clones. We also mine the source code to extract all the APIs that are been used by a developer. Through dynamic analysis we collect traces based on the system use.

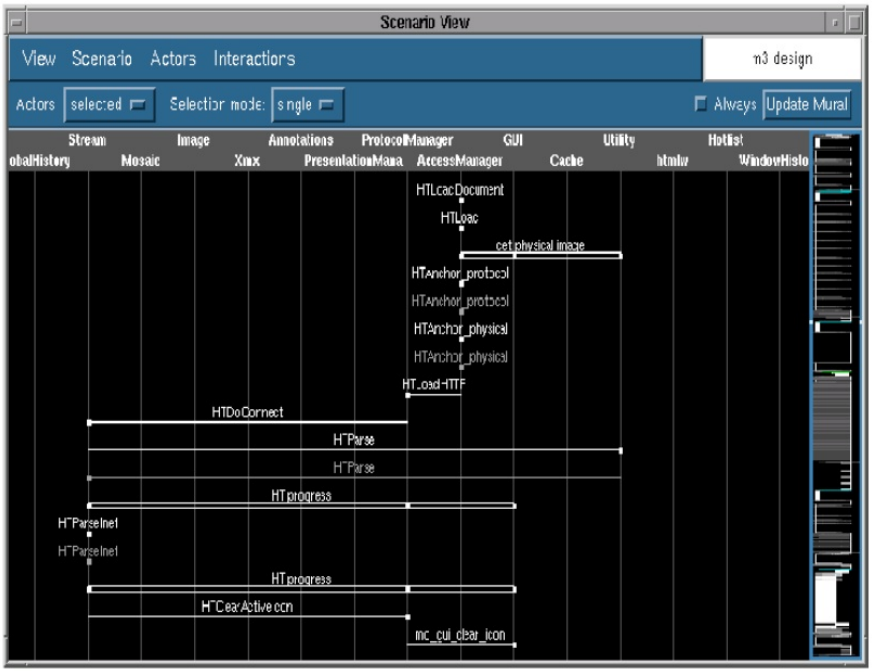

Figure 2.7: Diagram of ISVis (taken from [37, 38]) Showing Scenario View

2.7.2 ISVis(Interaction Scenario Visualizer)

ISVis [37, 38] is a visualization tool that provides analysis and comprehension of system at the architectural levels for C program. It has several graphical views that provide a platform to identify the components (major units) and the connectors (their interactions). The benefit of this tool is that, it too deploys both static and dynamic techniques and allows a user to get an overall picture of the system's architecture so that it is clear to them where in the system to bring about a change as a solution to any problem. Moreover, tasks such as design recovery, architecture localization, design or implementation validation, and re-engineering are also supported by this tool. The execution trace visualization is achieved using two types of diagrams: information mural and temporal message-flow diagram. The two diagrams are complementary to each other which after combining produces a scenario view (Figure 2.7). The information mural provide ways to create representation of large trace into concise form containing repeated patterns. The temporal message-flow diagram assists in viewing the detailed contents of the trace. In contrast our framework provides a platform to make the traces more manageable and meaningful by providing two level annotations. This annotation process segments the traces based on use. The uses are again segmented based on events. This assists the developer in understanding the context of use of specific or groups of method calls.

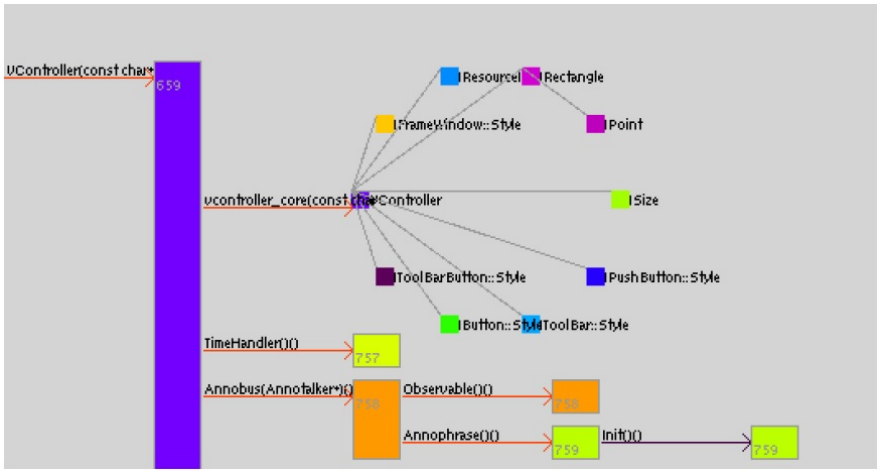

Figure 2.8: Diagram of Ovation (taken from[63]) Showing Communication Between Classes Using a Class Diagram

2.7.3 Ovation

This visualization tool [63] utilizes the execution traces to construct a tree based on view called the execution pattern view as shown in Figure 2.8. It allows the developers to view the program's execution pattern at different levels of abstraction. They can utilize the navigational, visual and analytical techniques of the tool to go over to the complex and lengthy traces. The main difference compared to Shimba [92] and ISVis [37] is that Ovation presents the view to the users based on the tree structure known as execution pattern view (Figure 2.8) which is less complicated than the UML sequence diagrams. It simplifies the visualization and its

unidirectionality in the axes and makes the representation more understandable and easier than interaction diagrams. With the view in hand the users can navigate through various features such as expand the subtree, zoom in and zoom out panels, manage repetitive sequence of method calls using colour code, select messages sent to particular object and so on. The matching criteria that are used to detect two patterns as equivalent are: identity, class identity, depth limiting, repetition and polymorphism. Our approach also incorporates the tree like structure to manage traces but in terms of context of use. On every click on a node a developer can understand what that node represents. Besides managing traces our approach also determines the behaviour of the trace nodes further using a set of metrics.

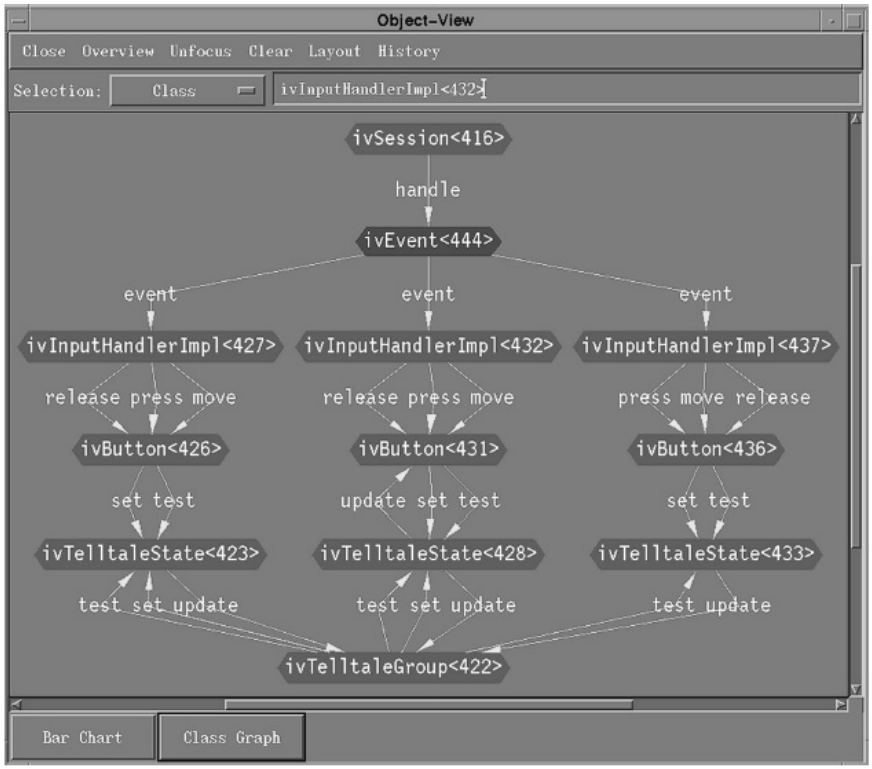

Figure 2.9: Diagram of Program Explorer (taken from [48]) Showing Interactions Using an Object Graph

2.7.4 Program Explorer

The tool [48] uses the combination of the static information and the execution data to produce a view of the system. It is mainly designed for the C++ language but it can be generalised for any object-oriented platform. It provides views for interaction among the objects and the classes using an object-oriented model and notation. Interactions among the objects are modelled using a three tuple directed graph called the interaction graph. The nodes of the graph represents objects and the connections between the graph show the interactions among the objects. One of the important features of Program Explorer is that, it can provide a basis for search space reduction using techniques such as merging, pruning and slicing. Merging of arcs

and methods can be achieved using object graphs as shown in Figure 2.9 fwhere multiple arcs of identical methods among objects are merged, and class graphs where objects of the same class are merged together. Pruning allows removal of unwanted nodes, arcs and activation path entering and going out of from the graphs, a process known as object pruning. Method pruning is identical to object pruning excepting the fact that only methods are removed rather than the objects while in class pruning the inheritance comes into play and any method in the inheritance hierarchy can be removed. On the other hand, slicing refers to keeping all the paths that are active in the program and dropping the rest that are infeasible. Our approach allows the developers to collapse and expand the traces all at once based on the developers need. It also implements a filtering option that permits analysis based on external and internal methods. This allows the developers to focus on only a single type of method and understand their behaviour. Moreover, the categorization feature provides further filtering based in groups of application programming interface and active clones.

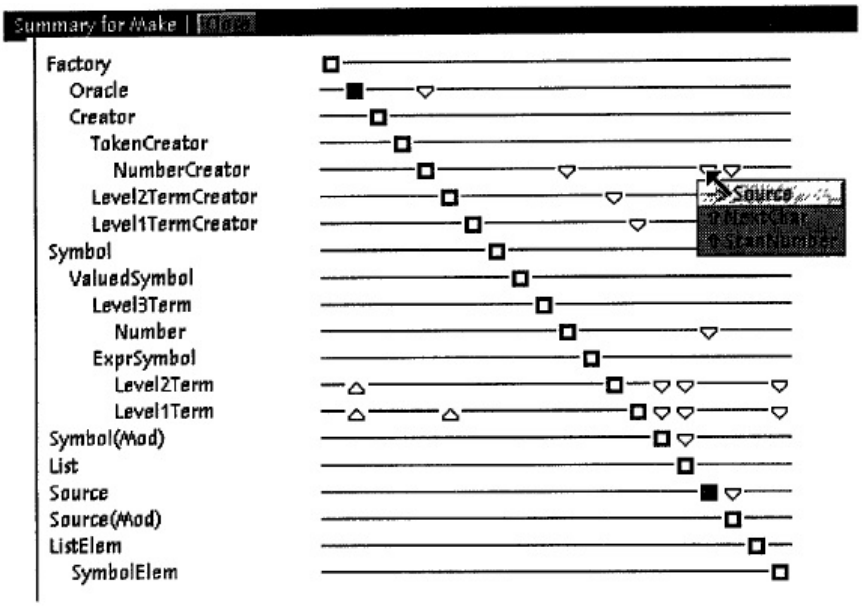

Figure 2.10: Diagram of SCENE (taken from [44]) Showing a Call Summary

2.7.5 SCENE (SCENario Environment)

As its name signifies SCENE produces scenario diagrams from the dynamic event traces of the object-oriented systems [44]. Scenario diagrams are a representation of the components such as objects, modules and processes and the interaction flows such as messages, procedure calls and events between them. The speciality of the tool is that it incorporates the concept of active text framework to provide hyper-text facilities. Moreover, the platform allows the users to look for other documents such as source code, class interfaces, class diagrams and call metrics besides the scenario usage. The tool itself is dedicated to focusing on problems allowing the users to navigate only those parts which they are interested in. Call compression, partitioning, projection and removal, and single-step mode are the techniques used to solve the problem. Call compression allows to collapse the internal invocations which on a mouse click expands them all as shown in Figure 2.10, and

partitioning divides the internal calls into smaller parts so that each part can fit into the screen and users can click on individual parts independently. Projection and removal offers the flexibility of choosing only a particular object and method involved in it while in single-step mode only one event is visible at a time and subsequent ones are revealed by clicking the next button. In our approach we provide more fine grained analysis by quantifying the traces using a taxonomy of metrics proposed by us. We also incorporate the UI thumbnail view and an enlarged snapshot view associated with every event of the trace. Clicking on the thumbnail provides enlarged picture of the UI. Each UI is also presented with a code viewer that displays the source code associated with it.

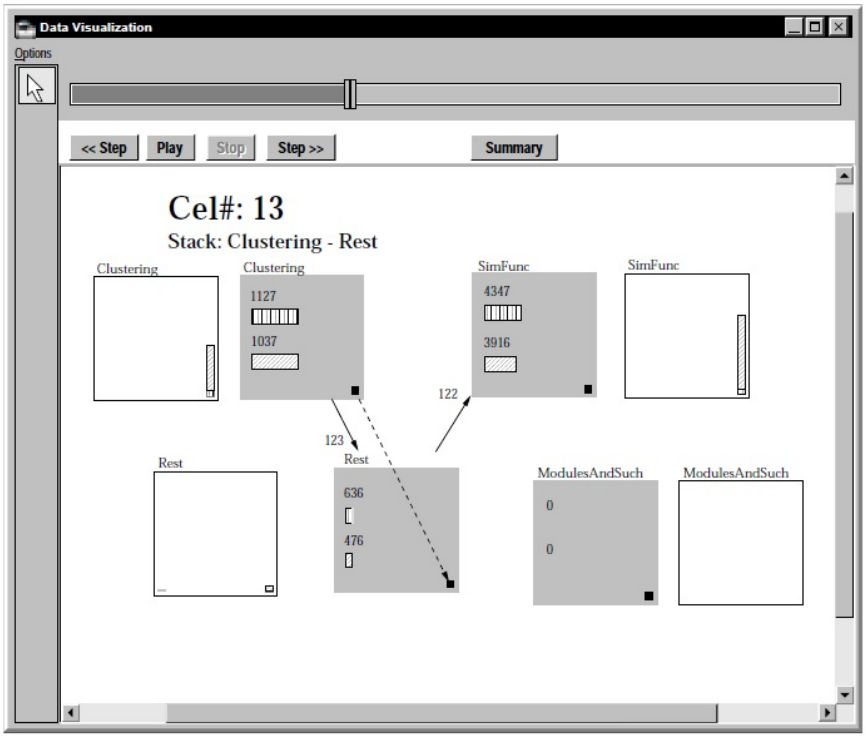

Figure 2.11: Diagram of AVID (taken from [94, 28]) Showing a Cel for Visualization

2.7.6 AVID(Architecture Visualization of Dynamics in Java Systems)

It is an off-line visualization tool [94, 28] that makes use of dynamic information to represent the operation of the object-oriented systems at the architectural level. The visualization of events can be viewed sequence by sequence in cels (Figure 2.11) which show dynamic information both at a certain point and the history of execution. The representation is done in the form of an annotated box containing methods invoked within it with arcs labelled to show the method invocation made between the source and destination classes. It also contains a summary view which shows all the interactions. The filtering technique is further reduced by applying sampling during which the user can select and analyse a specific sample of a trace. For instance, a section of a trace can be considered at a specific timestamp in the history. Our approach makes use of execution traces to represent a system's structure at the implementation level. We take into account the

method calls as the building blocks of the functionalities of a system and mine them to extract their behaviour. The thumbnail and image view panels permits the developers to see how a particular user interface behaves and helps them to map the method calls with the snaphots to understand the system's implementation.

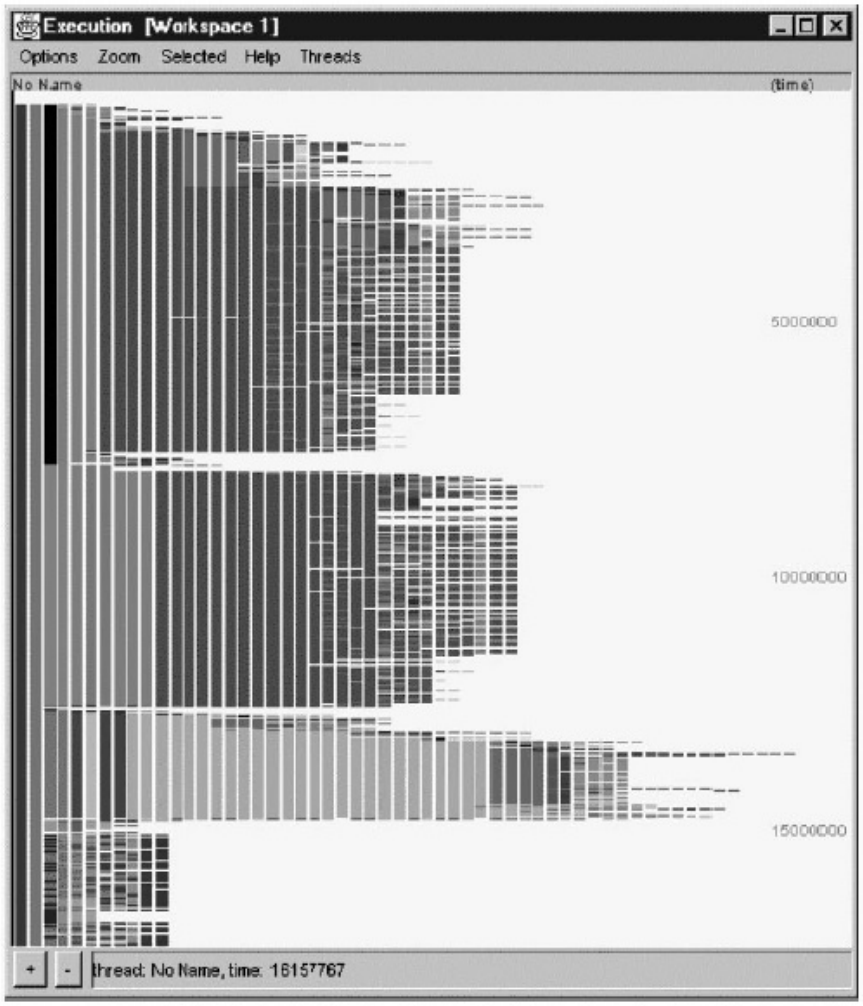

Figure 2.12: Diagram of JInsight (taken from [62]) Showing an Execution View

2.7.7 JInsight

JInsight [62] is a tool mainly designed for performance analysis, debugging and other tasks such thread and garbage collector activity, deadlocks and so on. It assists the users analysis through a number of different view layouts. The histogram view is used to detect the performance bottleneck in terms of resource consumption for classes, methods and instance usages. The hot spots are revealed shown in coloured rectangles. Reference pattern view tends to simplify the interactions between the objects by removing the information overhead. It uses the pattern matching algorithm to identify the recurring events in the object invocation and their relationships. Execution view as shown in Figure 2.12 provides the users with the program's sequence of calls

that may allow them to detect the hot spots and performance deadlock. The call tree view gives more precise view in numerical format such as total time taken for the execution of a particular task. A four dimensional model is used to represent an event that corresponds to object construction, destruction, method calls and return. To deal with the large overhead of information the concept of call frame is used which considers a set of events of communication patterns among objects as a one small unit. Our approach also allows analysis of traces not only through identifying features but also enhances the understanding of features through a set of filtering and categorization. It also provides the developers to gather more in-depth analysis of the features using a taxonomy of metrics. To manage the traces further the concept of active clones has been incorporated into our approach which reveals not only their context of use but also their dynamic behaviour.

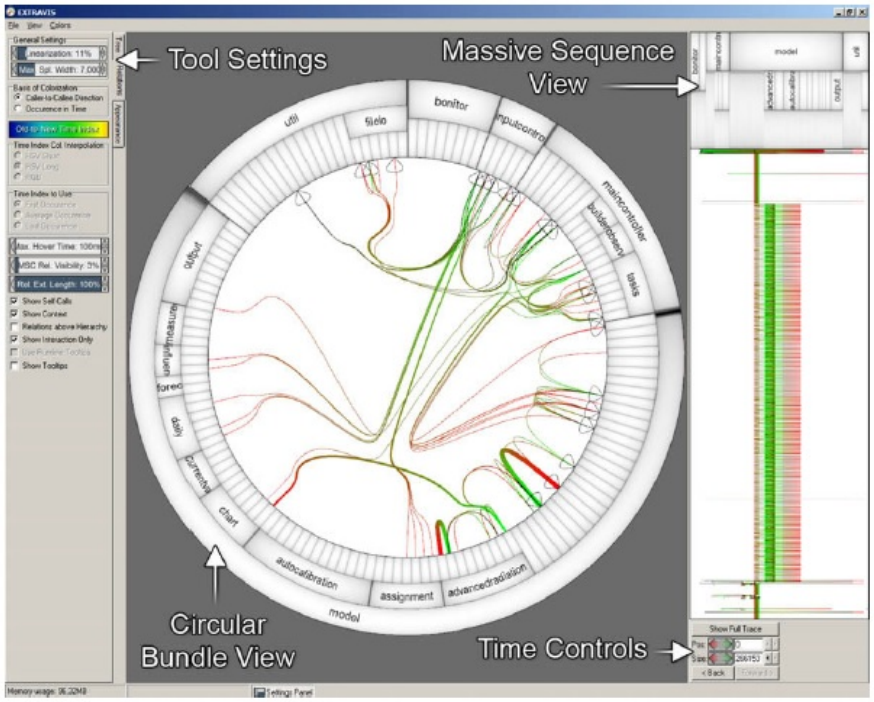

Figure 2.13: Diagram of Extravis (taken from [15]) Showing View of Execution Trace

2.7.8 Extravis

Extravis [15] is a highly scalable trace visualization tool that assists the user in program comprehension tasks such as trace exploration, feature location and top-down analysis with domain knowledge. It presents the users with two different types of views: circular bundle view and massive sequence view which is depicted in Figure 2.13. Circular bundle view provides view of the system's entities and their interrelationships among them. To provide an interactive and detailed navigation of the high level structural entities, Extravis offers features to collapse specific portions to have better view on it. It also provides visualization facility for traces corresponding to a specific snapshot in time. However, unlike the circular bundle view, the massive sequence presents the users with views of all the consecutive method calls in a chronological order. To

enhance the comprehension process the method calls are colour coded using a gradient. In addition to it, if any segment of execution trace needs closer examination the tool offers options for zooming into it. Above all, the synchronization between the two views provides the user to look into both of them while interacting with only one of the views. The thumbnail view, enlarged image view and the code viewer in our approach helps the developers understand the nature of the graphical user interface. It provides a linkage between the trace, the snapshots and the associated code all together in a single environment.

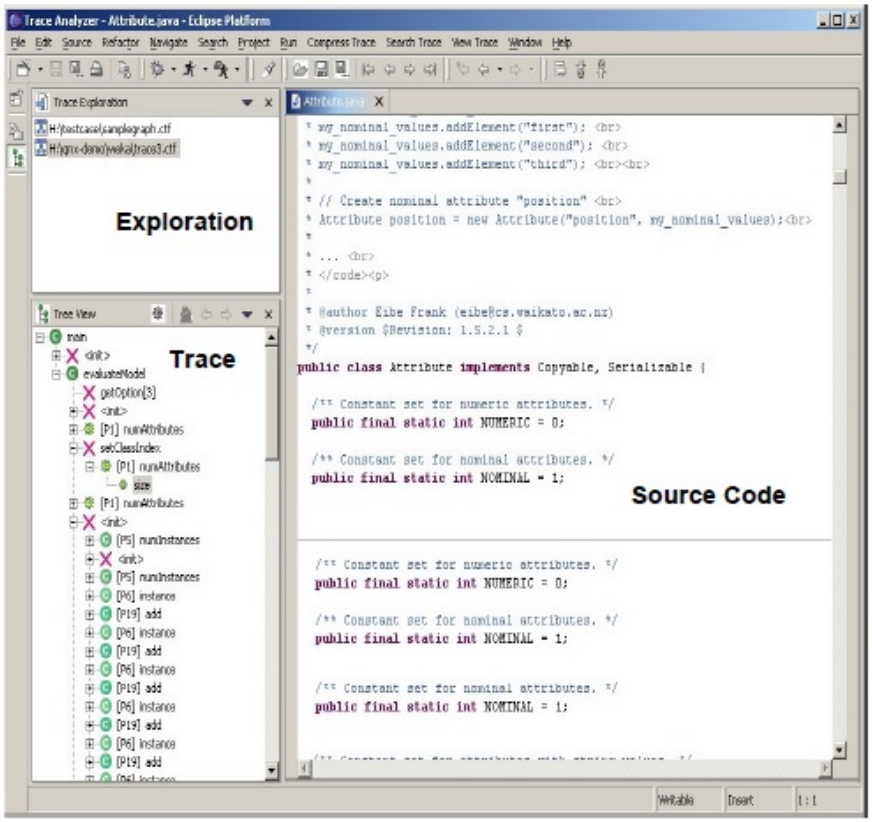

Figure 2.14: Diagram of SEAT (taken from [30]) Showing a GUI

2.7.9 SEAT(Software Exploration and Analysis Tool)

SEAT[30] is a trace exploration tool as shown in Figure 2.14 that assists the users to navigate through the traces and understand a system's implementation. The traces collected are represented using CTF (Compact Tree Format) which are displayed to the user using a set of visualization techniques in a tree-like structure. SEAT incorporates a number of different trace analysis techniques such as filtering techniques that allows the user to comprehend useful information. The graphical user interface is based on the Eclipse platform containing a multiple page trace editor along with a number of dependent editors. The trace is displayed in the trace editor while statistics and execution patterns are presented in the helping window. SEAT also

implements a number of metrics to quantify traces and offers feature to map traces with source code. It also supports features for searching of specific components using wild-cards. SEAT also provides provision for analysing execution pattens using colour coded techniques for distinguishing among different patterns. Our approach also displays the traces in a tree-like structure in a large main panel where the developers can analyse them by clicking on the nodes. Groups of method calls are clustered into events and groups of events are bundled into system use. This provides the context of use of the methods and can help in identifying the features.

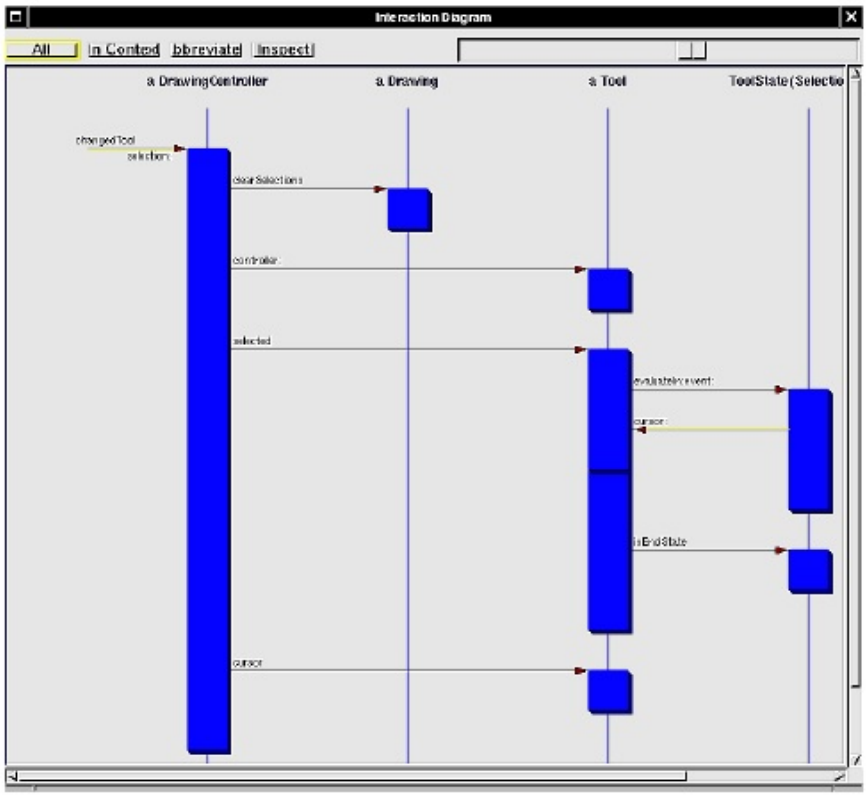

Figure 2.15: Diagram of Collaboration Browser (taken from [70]) Showing Interaction Between Objects

2.7.10 Collaboration Browser

Collaboration Browser [70] provides a platform for the users to visualize the interactions and collaborations (Figure 2.15). It deploys dynamic analysis and iterative approach to facilitate the design recovery and understand the dynamic behaviour through analysing the collaborations. The collaborations are formulated based on dynamic information, pattern matching and an iterative process. Dynamic information records the flow of control from one class to another in the form of execution traces while pattern matching detects the similar execution patterns in the traces and groups collaboration instances into collaboration patterns. To refine the analysis of collaboration further, an iterative process offers querying the dynamic information. The

tool provides the users with the options of dynamic information querying, editing of dynamic information through filtering and finally representing the result in the form of interaction diagrams. In our approach groups of method calls are represented in the form of call graph under events and system use.

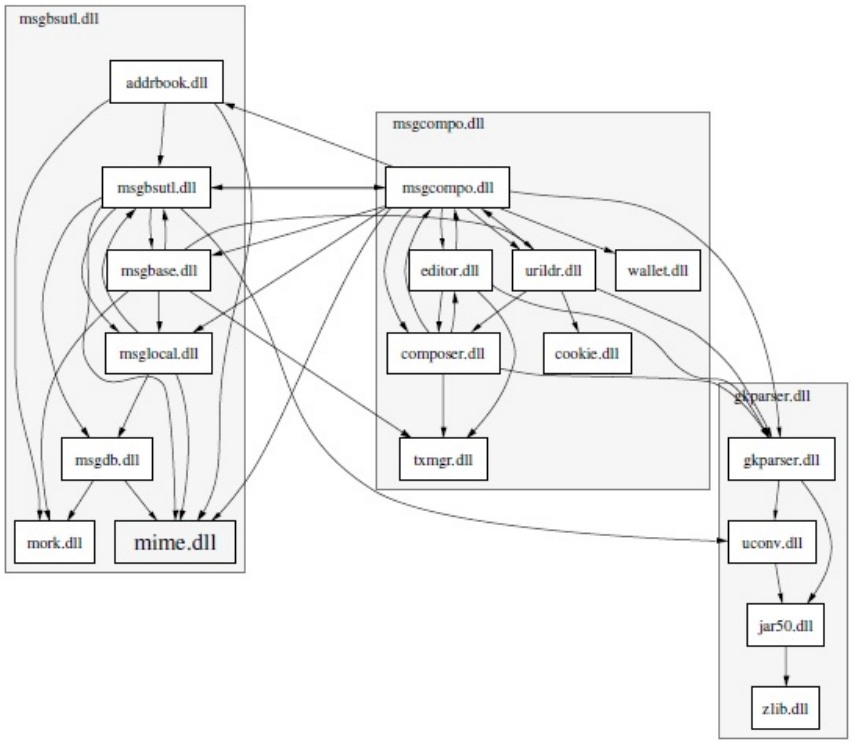

Figure 2.16: Diagram of Salah et al. (taken from [85]) Showing Module Interactions

2.7.11 Salah et al.

Salah et al. [85] proposed an approach for locating specific features in the source code. Their approach adapts dynamic analysis to reveal views of software system at different levels of abstraction. It relies solely on the use cases used to run the system. The execution traces which are obtained are further processed to locate the software features in the form of use cases, module interactions and class interactions as shown in Figure 2.16. Their architecture is composed of data collection, storing of data in the repository and visualization. The data collection module collects runtime data through code instrumentation using Microsoft C++ compiler's /Gh option. The data repository stores the program entities, relationships, and runtime events that are gathered from the subject system and the visualization represents data to the users in the form of views. Here graphs are used to display the visualizations where navigations among the different views can be achieved by selecting nodes that are used to represent the graphs. The views are constructed hierarchically where the topmost level represents the use case views, the second level indicates class relationships and the lowest level gives the interactions among the modules. In our approach we identify features by annotating groups of method calls under events and system use. The annotation process segments the traces so that they can be made more

manageable and at the same time make them more meaningful by qualifying them with their context of use. UI thumbnail, image snapshot and code view feature complements the feature comprehension task further. Navigation through the image snapshots can help the developers to compare the implementation of one UI from another.

2.8 Profiling Tools

Profiling is the technique used to identify system's performance, resource utilization and its behaviour.

Djprof [64] is a Java profiling tool implemented using AspectJ that can instrument code for the profiling of system as desired. The tool supports profiling such as heap usage (memory consumed), object lifetime (time between creation and garbage collection), wasted time (time between creation and first use) and time spent (time spent in each method). Hprof [50] is a command line profiling tool that is dedicated for heap and CPU profiling. It can be loaded dynamically with the JVM process at the command line and provides a textual or binary format output. The users can select from various heap and CPU profiling features in it. The implementation is based on JVMPI [50] (Java Virtual Machine Profiler Interface) which is a bi-directional call interface between the JVM and the profiler agent. Gprof [64] on the other hand, provides a call graph view of the system and utilises gcc to instrument the code. It produces result in two different formats: flat format in which the time of execution of each function call is shown and the call graphs show the results in hierarchical form with times spent in each parent and child methods. Gilk [65] which is also an instrumentation tool for Linux Kernel allows addition of instrumentation code at runtime. It is achieved using code splicing where a branch instruction is added at the instrumentation point. The overwritten code is relocated into a code patch which is targeted by the code splice. Ltrace and strace are linux based tools which are used to trace library system calls of a process. The strace command can collect traces either by running a new command or attaching itself to already running program. Similarly, ltrace is used to collect trace for dynamic library calls to provide method timings and number of executions they have executed.

JRat (Java Runtime Analysis Toolkit)[2] is an open source profiler that measures performance of open source java systems. It instruments a source code and monitors the behaviour of the running application and profiles performance measurements. The performance output of a system can be viewed and analysed in a graphical user interface facilitated with a number of view controls. JMP (Java Memory Profiler)[3] is used to profile execution of java system and provide object usage and method timing. JMP has the capability to provide the user to analyse heap in order to detect memory leaks. It also provides statistics for method timing, number of calls and time take in methods called. The call graph generated also supports the caller and callee representation. On the other hand, JIP (Java Interactive Profiler)[4] is similar to hprof and is shipped with JDK. Unlike hprof that uses JVMPI, it uses aspects to instrument code for every method of every

[2]http://jrat.sourceforge.net/
[3]http://www.khelekore.org/jmp/
[4]http://jiprof.sourceforge.net/

class. It provides a call graph and allows the users to filter classes by class names and packages. JProfiler[5] is a tool with wide range of features to support profiling and monitoring of java systems. Its UI allows the users to track performance bottleneck, analyse issues with threads and find memory leaks in the system. It deploys dynamic slicing technique to analyse execution traces for program debugging and understanding. In our approach we adapted aspect oriented programming to instrument code and gather run time data in the form of execution traces. We also mine source code to extract all the methods, API used and use NiCad [77, 13, 12] to detect clones. Our approach allows the developers to analyse the run time information through a taxonomy of metrics at various levels such as system use, events and methods. Our approach also allows the developers to analyse the behaviour of different graphical user interfaces of a system used during the runtime.

2.9 Source Code Feature Analysis

Advances in software re-engineering and program comprehension have also been achieved using dynamic analysis. Research on execution trace based analysis is used to manage, comprehend, and quantify large amounts of method traces obtained during the program execution. Sutherland and Schneider [89] proposed an approach that can aid in the maintenance of interactive software systems with user interface traces. In their study they presented an approach for navigating the architecture and source code of interactive software applications based on actions that occur in the user interface. Contributions by Hamou-Lhadj and Lethbridge [26, 27], Eisenbart et al. [17], Asadi et al. [2], Zaidman and Demeyer [98], Zaidman et al. [99], Lo et al. [51], Voigt et al. [93], Thomas et al. [25] all aim to make the trace data more manageable by mining important features and concepts from them for program comprehension.

Fischer et al. [22] proposed a technique to study the evolution of systems of different versions using the execution trace. They developed a tool called EvoTrace that uses probe insertion as a method of code instrumentation. It can track high level module views and report findings in three kinds of visualizations. Safyallah and Sartipi [81] took feature specific task scenarios into consideration and proposed a new methodology to identify groups of core functions implementing software features. Lo et al. [51] used execution traces to mine temporal information of arbitrary length to reveal the invariants for assisting the developers to understand the downstream applications such as verifications. In our approach we perform both trace management and feature identification. We deploy an intermediate layer called annotation during the program instrumentation process. This help in managing traces by segmenting them into modules that signifies the context of use of the groups of method calls. We also cluster the traces based on method types(internal or external) and further classify them into groups of API and active clones. Developer can analyse the runtime data based on groups or method types. To enhance the feature identification the thumbnail view, image snapshots and code viewer helps to support the feature identification easily.

[5]http://www.ej-technologies.com/products/jprofiler/overview.html

2.10 Code Clones

This section presents a brief overview of the concept of clones, terminologies and state of research. Clones are also a part of this dissertation and we have applied its concept in the study as well.

2.10.1 Definition

Code clones are fragments of code that are identical or similar to each other. A code fragment is a set of lines of code that starts and ends at a specific positions in the code. It can be represented using a three tuple notation such as $cf = (f, s, e)$ where f is the filename, s is the start line and e is the end line. It can be any sequence of code in a file i.e., it could be a either a method or a block. Two code fragments that are similar to each other form a clone pair and a group of clone fragments that are similar to each other are clustered together into clone a class. The similarity between code fragments are based on two kinds of measures: textual similarity or functional similarity. Textual similarity refers to the notion of the representation of text in code fragment and are of three types: Type 1, Type 2 and Type 3.

Type 1 clones which are also known as exact clones are exactly similar to each other in terms of textual representation except variations in comments, whitespaces and layout. Therefore, the clone class into which they fall into is known as Type 1 clone class which is shown in Figure 2.17.

Clone Type	Fragment1	Fragment2
Type1	**int a=0, b=1;sum=0;** **while (a!=100){** **b = b + a; a+=1;** **if (b == 25) {** **sum[a] = b;** **break;** **}** **else** **continue;** **}**	**int a=0, b=1;sum=0;** **while (a!=100){** **b =b+a; a+=1;** **if (b == 25) {** **sum[a] = b;** **break;** **}** **else** **continue;** **}**

Figure 2.17: Type 1 Clone Fragments

Type 2 clones are fragments that are exactly similar in structure but with variations in identifier names, comments, types, layouts and literals. In the Figure 2.18 fragment1 and fragment2 are Type 2 clones with identifiers in fragment2 represented in red colour have been renamed.

Type 3 clones are the ones in which the fragments are modified such that more statements are added, deleted or changed further in addition to variations in comments, identifiers, layouts, literals, types and whitespaces. Type 2 and Type 3 clones fragments are often known as near-miss clones. From Figure 2.19 we can observe that in fragment2 besides variations in identifier names, layouts, whitespace and comments a new line has been added which is represented in blue colour.

Clone Type	Fragment1	Fragment2

Type2	int a=0, b=1;sum=0;
	while (a!=100){
	b = b + a; a+=1;
	if (b == 25){
	sum[a] = b;
	break;
	}
	else
	continue;
	}
		int c=0, d=1;sum=0;
		while (c!=100){
		d=d+c; c+=1;
		if (d == 25){
		sum[c] = d;
		break;
		}
		else
		continue;
		}

Figure 2.18: Type 2 Clone Fragments

Clone Type	Fragment1	Fragment2

Type3	int a=0, b=1;sum=0;
	while (a!=100){
	b = b + a; a+=1;
	if (b == 25){
	sum[a] = b;
	break;
	}
	else
	continue;
	}
		int c=0, d=1;sum=0;
		while (c!=100)
		{
		d=d+c;
		c+=1;
		if (d == 25){
		sum[c] = d;
		printf (sum[d]); //prints array
		break;
		}
		else
		continue;
		}

Figure 2.19: Type 3 Clone Fragments

Type 4 clones are classified based on the functionality of the code fragments. Therefore, a Type 4 code fragments exhibit identical functionality but possess different textual representations. Type 4 clones are also known as semantic clones. Two code fragments that calculates a factorial of a number but are represented one using a loop and another using recursion is an instance of Type 4 clones as shown in Figure 2.20.

Clone Type	Fragment1	Fragment2

Type4	int fact (int n) {
	int a=0;
	for(int i=1; i<=n; i++){
	a=a + i;
	}
	return a;
	}
		int fact (int n) {
		if(n==0)
		return 0;
		else
		return n + fact (n-1);
		}

Figure 2.20: Type 4 Clone Fragments

2.10.2 Reasons for Code Clones

Code clones do not occur in systems by themselves and cloning through copy-paste is a common practice undertaken by the developers. The amount of clones in a system may vary depending upon the domain and the origin of software system. One of the reasons for code cloning is the copy paste activity done by the developers in order to speed up the development process for meeting the deadline. There are several factors as well identified by the researchers and a classification of them into four categories has been proposed by Roy and Cordy [74, 75]. The four classifications are as follows:

1. **Development Strategy:** Code clones are introduced into a system due to various development strategies such as copy-paste due to code reuse, design, logic and functionality. Reusing code through copy-paste is the fastest and the simplest way for reusing the syntactic and the semantic constructs. On the other hand, design, logic and functionalities can be reused when there are similar solutions available. Cloning is also dependent on the programming approaches such as merging of two systems having similar functions, using generative programming approaches where the tools use templates to implement similar functionalities or delay caused by the programmers in restructuring the code.

2. **Maintenance Benefit:** Clones are often encouraged in the system to assist the maintenance activities. Modifying a system by adding new functionalities requires implementation and then testing to ensure proper functioning of the system. Once new functionalities are added, then there exists a risk for errors in the code fragments which may make the system vulnerable. So in that case developers are usually asked to copy paste code fragments that perform similar functionalities. Cloning also helps in providing a way to keep the architecture clean and comprehensible. Since clones are also independent of each other, evolution can take place at great speed and help in accelerating the maintenance process.

3. **Overcoming Underlying Limitations:** There are several limitations regarding the languages and the programmers for which code clones are introduced in the system. From programmer's point of view, factors such as a developer may have difficulty in understanding the system, a specific time constraint has been assigned, measurement method (lines of code) used to determine a programmer's performance or developer's lack of knowledge on a problem domain and lack of ownership of the reused code all encourage cloning to take place. However, developers also face problems where language limitation do not permit them to write reusable codes and even if they do so it might be error-prone and detrimental to the existing system.

4. **Cloning by Accident:** In this case cloning takes place when two programmers working on the same logic coincidently end up with similar implementations independent of each other. It might also happen that developers are influenced by their memory in which they are unintentionally tempted to repeat common solutions to problems with common patterns in their memory for that particular problem. Such cases give rise to clones that the developers are not aware of it.

2.10.3 Harmfulness of Code Clones

Although code clones are beneficial in terms of development strategy, maintenance benefits, and overcoming underlying limitations they have their own disadvantages as well. Since cloning is inevitable, large systems do contain clones which have severe effects on the quality of the systems. Some of the harmfulness of code clone in the system can be described as follows:

a. **Bug Propagation:** Although the practice of copy-paste assists in accelerating development process, it tends to put the system into risk as well. For instance, if a code fragment contains a bug and it is being copied into various locations then the bug will also propagate with the copied code in every direction.

b. **Inconsistent Code Modification:** Code fragments often need to be modified later as a part of maintenance activity in order to fix inherent bugs or to add new functionalities. If a system contains a lot of clones then each cloned fragments needs to be modified to establish the consistency. Otherwise the unchanged code fragment will contain bugs making the system inconsistent or even cause malfunctioning.

c. **Increased Maintenance Effort:** One of the major impact of code cloning is code inflation. Copy-paste activity not only increases the size of the codebase but also enhances the maintenance effort. A change in one fragment must also be propagated to other fragments which requires additional effort to understand the system and thereby apply the similar set of changes to them.

d. **Increased Resource Consumption:** Since code cloning increases the size of codebase the requirement of storage grows as well. Although this does not cause any problem for some domains, it also poses severe impacts on devices such as telecommunication switches where hardware upgrades have to be done in accordance with the cloning in software. Moreover, cloning also impacts compilation process as more code needs to be translated.

e. **Impact on Design:** Cloning also impacts on design where it might introduce bad design, poor inheritance structure or abstraction. As a result code reuse becomes difficult in future. When the system gets incorporated with poor design, then maintainability of a system becomes hard as well.

2.10.4 Code Clone Evolution

Software evolves due to various reasons such as bug fixing, increasing reliability, improving performance, and adding new features to meet customers' requirements. Clone evolution analysis is concerned with understanding not only the changes of clones, but also the causes and the effects of those changes. Results of such analysis can help us to infer about cloning, improve software processes for managing clones, support developers in copy-paste programming practices and allow maintenance engineers to make informed decisions about removing clones.

The evolution of a clone fragment over versions with respect to other clone fragments in a clone class is known as clone genealogy [83]. Six change patterns are identified based on the changes of code snippet and the number of clone fragments in the same clone class in two consecutive versions. The change patterns are as follows:

Let CCi be a clone class in revision Ri is mapped with CC_{i+1} in revision $R_(i+1)$ by a clone genealogy extractor. Now the change patterns can be described as follows:

a. Same: The clone fragments in CCi are present in CC_{i+1} and no additional clone fragment is added in CC_{i+1}.

b. Add: One or more clone fragments are added to CC_{i+1} that were not in CCi.

c. Delete: One or more clone fragments of CCi do not appear in CC_{i+1}.

d. Static: The clone fragments in CC_{i+1} that were part of CCi, have not changed.

e. Consistent Changes: All fragments in CCi have been changed consistently, thus, all of them are again part of CCi_{i+1} in Ri. However, a clone class may disappear after being changed consistently, if fragments become smaller than the minimum clone length of the clone detection tool.

f. Inconsistent Changes: All clone fragments in CCi have not been changed consistently. Here we should note that as lines can be added or deleted from Type 3 clones, all the clone fragments of a particular clone class could still form the same clone class in the next revision even if one or more fragments of that class have been changed inconsistently. The dissimilarity between clone fragments in a clone class depends on the heuristics or similarity threshold of clone detection tools.

2.11 Application Programming Interface (API)

2.11.1 What is an API?

An Application Programming Interface (API)[23] is a set of commands, functions, and protocols which a programmer uses to build an application. It provides a short cut way for the programmers to use the functionalities of the system instead of developing the same functions from scratch. It serves as an interface between two software programs and facilitates interaction between the software unlike a user interface which allows interaction between humans and computer. In general an API is a software to software interface. The vital role of an API is creating communication between the systems so that the users can get the required functionalities and information from the system. An API can be used in one of two ways in an object-oriented system: method invocation or inheritance.

```
public class NewClass {

    public static void main(String args[]){

        Vector v = new Vector();
        v.elementAt(5);
        v.remove(10);
    }

}
```

Figure 2.21: Method Invocation

```
class MyJFrame extends JFrame implements ActionListener {
  JButton b1, b2, b3;
  SimpleTree panel;
  MyJFrame(String s) throws IOException {
    super(s);
    setForeground(Color.black);
    setBackground(Color.lightGray);
    panel = new SimpleTree();
    collapseAll(panel.tree);
    getContentPane().add(panel,"Center");
    ...
  }
}
```

Figure 2.22: API Inheritance

API use via Method Invocation

In this technique the API methods are called directly or through object instantiation. The Figure 2.21 illustrates the method. For example in the Figure v is an object of the class type Vector and the methods elementAt() and remove() in the class Vector have been called via invocation.

API Use via Inheritance

Figure 2.22 illustrates how an API can be used by inheritance. By definition, inheritance is a way of reusing code of existing objects. It is implemented by using the keyword extends. The inherited class is called the superclass and the inheriting class is called the subclass. In the figure the class MyJFrame inherits the JFrame class using the extends keyword. The setForeground(), setBackground() and getContentPane.add() are the methods that are inherited from the JFrame class and thus they are used in the MyJFrame class without object instantiation.

2.11.2 Benefits of an API

There are a number of benefits provided by using an API. They are:

- Provides an interface to the programmer to access certain functionality without requiring them to write the code from the scratch.
- Allows programmers to save time through code reuse.
- APIs do not expose the underlying code to the programmer. As a result changes to the other parts of the program do not affect the code associated with the API.
- APIs provide a medium for accessing certain underlying operating system functionality that are not accessible without them. Such functions include access to device drivers, handling interrupt service routines of the operating system, accessing the registry and so on.

2.11.3 API Studies

With increase in availability of data and growing demand of service, software development has become a challenging area. At the same time maintenance of software is also getting more of a noticeable concern. Once software is developed and ready for commercial use it needs to be maintained at regular intervals. The maintenance task includes software upgrades, addition of new functionalities and fixing of bugs. Typically the larger software is, the greater the maintenance cost will be. Consequently, the question is how large and complex software could be. The answer to this question depends on various factors ranging from the type of the system, customer requirements, platform and the implementation style. Previously it was obvious that developing a system required considerable amount of coding from the scratch. However, with the rapid growth of software development technology and sophisticated programming platforms the task of coding has become very easy and simple. The programmers are able to quickly adapt themselves to the platform and carry on with the development process. One of the major breakthroughs in this technique is the support of interfaces allowing the developers to do coding through code reuse. Such reuse is provided by built-in interfaces known as an API (Application Programming Interface). With the rapid progress of software engineering reusable components, frameworks, APIs and libraries have been developed. As a result the process of system development has shifted considerably where the programmers can keep the size and complexity of the program small and simple without having to sacrifice functionality. Studies [54] show that the Java Standard API consists of about more than 3000 classes and 20000 methods. The same study has revealed that only about 50% of the classes in the Standard API are used at all, and around 21% of the methods are used.

Large APIs [86] like Microsoft's .NET Framework or the Java APIs have grown to thousands of classes with tens of thousands of methods, and grow larger with each successive release. Previous studies show that Microsoft has created and supported many different application programming interfaces (APIs) that are in

wide use today. The .NET framework APIs alone include more than 140,000 methods and property fields and are shared by a collection of programming languages including C#, VB.NET and C++.

The usability of an API is very important for the productivity of the programmers and software coding. The usability of an API is of much talked about issue nowadays. Although there are enormous numbers of APIs and libraries packaged within programming language software with diverse functionalities there is still a need for proper documentation for the programmers to understand and use them accurately. The programming skill, efficiency and performance of software depends how optimally code is used to achieve the desired functionalities. There comes the usability issue of an API. Knowing how well to use APIs depends how well they are documented. Little research has been attempted to address the usability of an API in terms of difficulty and the extent of the number of APIs used in the development of software.

A number of studies have been conducted on API usability which can be classified as research in design issues, tool support, API migration, and API usability measurement techniques. Stylos et al. [88] in their research concentrated on API design decisions. They focused on defining what are the factors that must be considered in designing a good and powerful API. They also proposed a mapping of API design decision space with API quality attributes and identified different metrics for arriving at the decision. In another research, Stylos et al. [87] raised the issue of combining the initial and the new users requirements for redesigning APIs. They performed a case study on SAP BRPlus which is a business rule engine. They did a usability evaluation on a user-centric design of an API wrapper to assess the value of addressing specific use cases by the application-level developers. Similarly they also carried a comparative study on the usability of API design by considering the programmers preferences on using APIs with parameters or without parameters. They concluded that more preference was towards APIs without parameters instead of the those with parameters.

Bartolumoi et al. [6] studied API migration between two different XML APIs. They applied the process of API migration on two different Java platform XML APIs. These APIs were further investigated to find the differences with measurement and identify of classification based on usability.

Kawrykow et al. [41] did extensive research on the effective use of APIs. They identified the fact that though APIs are intended for simplicity and code reuse, however in some cases the programmers are not using the APIs built in the software package. They rather try to re-invent the code and use them in their development. Such scenarios are defined as inefficient API usage. To address this issue they have created a tool which can automatically detect such inefficient API patterns. On applying their tool on Java projects they found that around 4000 cases were among the victims of inefficient API usage that required improvement.

Dekel et al. [16] did research on improving the documentation to convey the hidden information in it to the interested reader which may get ignored within the long text. They have built an eclipse plug-in called eMoose to highlight those hidden directives.

Feilkas et al. [21] worked on identifying whether an API client complies with the API developers assumptions of the domain abstraction. They developed a framework for expressing the assumptions that restricts the clients in not complying with the expectations of the provider.

Arnold [1] and Henning [33] pointed out that creating a useful API depends on API design and human factors. Bloch [8] and McLellan [55] et al. suggested a number of guidelines for building a good and usable API. On the other hand, Ellis et al. [18] conducted a comparative study on the usability of factory patterns and the constructors on the instantiation of objects. They found that factory patterns are harmful to API usability and concluded that more time is required by the user to construct objects with using a factory pattern.

Measuring API usability is also of great importance and Clarke and Becker [11] defined 12 cognitive dimensions for measuring usability of an API while Bore et al. [9] contributed in proposing seven different measures in order to profile API usability. Wu et al. [95] proposed a tool called CoDecent which can automatically link API documents and create a diagram for the users to understand the use of an API. MAPO is another tool developed by Xie et al. [35] which mines API usage patterns while Stratcona [96] is another specialized tool that can aid in example matching between API source code and the source code repository containing the API. Catch-up [32] and Diff Catch-up [97] provide support to the users during the code change caused during the evolution of the APIs contained in the code.

Chapter 3

Framework

3.1 Introduction

Program comprehension is one of the vital activities in software maintenance and it is a part of the development process that enables knowledge gathering about software system. A system when in need of maintenance must be accompanied by a good program comprehension technique. Proper maintenance depends on how well a maintenance engineer can understand the existing system. Software development, maintenance and program comprehension are inter-related. Program comprehension by nature is a very difficult activity because of the rapid evolution of software due to frequent changes in customer requirements. Therefore, it demands attention for development of easy, simple, automated and informative comprehension techniques.

In this thesis, we present a framework that supports developers and provides guidance for program comprehension. The framework consists of an interface which will allow the developers/maintenance engineers to navigate through the entities of the system and gather their usage in terms of the context. In this regard our framework as shown in Figure 3.1 consists of three phases as follows:

P1. **Source Code Mining:** This is the initial phase of our study. After selecting a system, the source code is mined using static and dynamic analyses for collection of data. Static analysis extracts method, clone and API data from the source code. and the dynamic analysis collects runtime data of the system using a set of inputs.

P2. **Trace Collection and Mining:** Dynamic analysis is used to instrument the source code and the testing framework is used to collect run time data. Once the data are collected, they are mined for clustering, categorizing and calculating behavioural information using a set of proposed metrics.

P3. **Output:** Once the results are computed they are presented to the user in a viewable format with user interface where they can navigate using controls according to their need and choice of analysis.

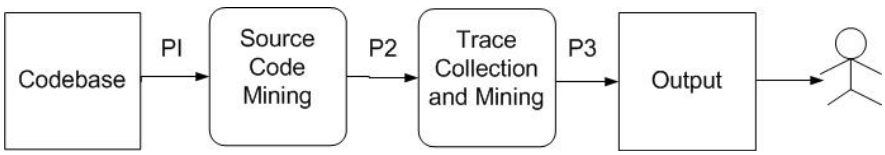

Figure 3.1: Overview of the Study

3.2 Terminology

This section provides a set of definitions related to the users using a system.

3.2.1 Use

A use is a set containing a number of tests. For instance, a use such as "draw object" can include a set of tests such as drawing a rectangle, a triangle, and a square. The tests together are termed a drawing test suite for the use "draw object". Each system use is associated with a number of user activities.

3.2.2 Use Set

A use set is a set containing a number of system uses. For instance, a "draw rectangle" use can include uses such as creating a rectangle, adjusting the shape, filling the rectangle with colour, and modifying the line width. Similarly, another use called "draw a circle" can include uses creating a circle, adjusting the shape, filling the circle with colour, and modifying the line width. Together the uses "draw rectangle" and "draw circle" belong to a set and can be termed as "Manage object" use set.

3.2.3 User Activity

A user activity is referred to as sequences of interactions performed by a user on the interface while performing a particular use of a system. For instance, user activities on a interface during drawing a rectangle may involve events such as mouse actions, keyboard actions, display updates, thread signals and so on.

3.3 Testing Framework

Our interest is in studying interactive systems with graphical user interfaces (GUIs) and so we developed a testing framework shown in Figure 3.2 that accommodates tests corresponding to a user interacting with the system. Four layers are identified in the framework: the test layer, the user interface layer, the functional layer and the source code layer. The framework connects user activities defined in use sets and tests to the source code. As a result, a system's use can be visualized as a sequence of user interactions within the interface carried out by a user in order to execute a particular feature. Each individual interaction is associated with a number of user events. For instance, mouse actions, keyboard actions, display updates, thread signals and so on are categorized as user interface (UI) events. Such UI events generated during the use of a system are denoted in Figure 3.2 as *Event* i where $i = 1, 2, .., n$. Each individual event in turn corresponds to sub-programs that handle the event, denoted in the Figure 3.2 as Si. A sub-program may be executed by more than one test.

A trace is a sequence of method invocations executed during a particular test. The methods are invoked in response to the user activities. An execution trace can be further processed to mine meaningful resource

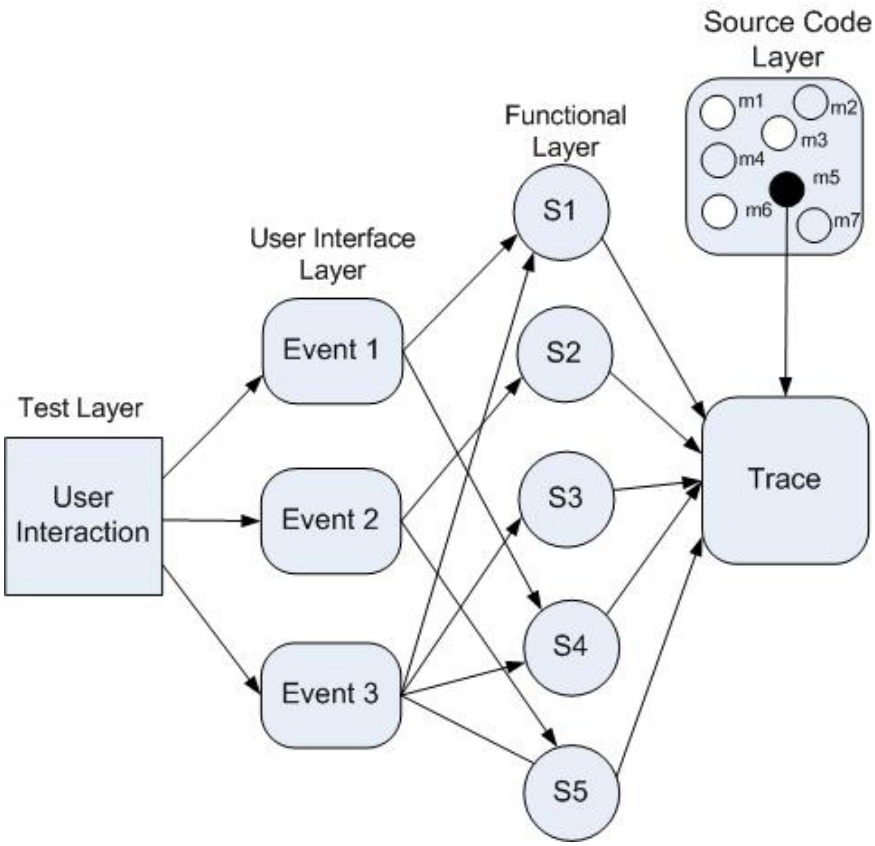

Figure 3.2: Testing Framework

information, such as CPU and memory usage. We are also able to segment the execution trace by user interface event to help manage the size and complexity of the trace data.

Our study in this thesis adapts a hybrid approach that blends both the static and the dynamic analyses. From the static point of view we are deploying clone and method detection and API extraction from the source code, while using dynamic analysis we are instrumenting the source code to extract active methods from the trace. Now, let us assume that Z is a set of all the methods in the systems and T be the set of all the methods that are invoked in the trace during the testing phase. So, the relationship between Z and T can be inferred such that $T \subset Z$ where $Z = \{m_1, m_2, m_3, ..., m_n\}$ and $m_i = i$th method of the codebase. Similarly T is a set of methods in trace such that $T = \{m_1^t, m_2^t, m_3^t, ..., m_n^t\}$ where m_t^i is an ith invoked method in trace t, for instance m_5 in Figure 3.2, also known as active methods.

3.4 Analysis Framework

In this section we will describe the framework for analysing a software system. Our framework is a blend of both static and dynamic analyses as shown in the Figure 3.4. It incorporates four phases: clone and method detection, code instrumentation, API extraction, and finally trace mining. Each phases are described in detailed as follows:

3.4.1 Static Analysis

Static analysis [19] is useful for mining specific information from a codebase without executing the software and it has been used for detecting clones in this thesis. An advantage of using a static technique is that we can often obtain information from the source code even if there is incomplete code. Furthermore, it is neither necessary to run the software nor does the software needs to be executable. The static analysis phase in our study is divided into two sub-processes: (a) Clone and Method Detection and (b) API Extraction which are as follows:

(1). Clone and Method Detection

In this phase we use a static analysis tool, the NiCad clone detector [77, 13, 12], to detect clones in the codebase. NiCad has been shown to be highly accurate with respect to both precision [76, 77] and recall [76, 79, 90] in the detection of copy/pasted near-miss clones. In our study we consider all non-empty methods, and blocks of at least five lines in pretty-printed format. We then use size-sensitive UPI (Unique Percentage of Items) thresholds [77] to find exact and near-miss clones. For example, if the UPI threshold (UPIT) is 0%, we detect only exact clones; if UPIT is 10%, we detect two code fragments as clones if at least 90% of the pretty-printed text lines are the same (i.e., if they are at most 10% different). Besides detecting clones, NiCad [77] also provides us with a set of methods and code blocks corresponding to them extracted from the codebase. Each code block is the portion of the code contained within the starting and ending curly braces. Once the clones are detected, clone metrics are calculated using the tool VisCad [3]. The clone metrics will further help in the trace mining process to derive the active clone metrics. Similarly in this phase, system metrics are also obtained using a line of code calculator called cloc[1]. Figure 3.3 shows a snapshot of the NiCad's XML file format for storing clone information. Here every clone fragment is enclosed within a class with a class id, number of lines, and number of fragments while each method is identified using a file name, starting and ending line number. Each clone fragment belongs to a particular types of classes such as Type 1, Type 2 and Type 3.

(2). API Extraction

In the second part of the static analysis we are considering all the APIs that have been used in the system. By API we mean all the external method calls that have been utilised by a developer during the implementation. Therefore, external APIs are those method calls belonging to a particular class and package shipped with the IDE from industry. The external method calls are accessed by declaring the packages in the import list at the beginning of the source code file. We then further parse each file of the codebase to extract the import lists from them. They are used to determine the external APIs, i.e. packages, classes and methods that

[1] cloc.sourceforge.net

```
<clones>
<systeminfo  system="JhotDraw7.0.6" granularity="functions" threshold="30%" minlines="5" maxlines="500"/>
<classinfo npcs="3260" nclones="375" nfragments="600" npairs="688" nclasses="225"/>
<runinfo ncompares="311676" cputime="220000"/>

<class id="1" nlines="80" nfragments="3">
<source file="C:/Systems/JHotDraw7.0.6/JHotDraw7.0.6/src/org/jhotdraw/xml/NanoXMLDOMOutput.java" startline="194" endline="272" pcid="847"
></source>
<source file="C:/Systems/JHotDraw7.0.6/JHotDraw7.0.6/src/org/jhotdraw/xml/NanoXMLLiteDOMOutput.java" startline="182" endline="260" pcid=
"890"></source>
<source file="C:/Systems/JHotDraw7.0.6/JHotDraw7.0.6/src/org/jhotdraw/xml/JavaxDOMOutput.java" startline="196" endline="268" pcid="868"
></source>
</class>

<class id="2" nlines="76" nfragments="2">
<source file="C:/Systems/JHotDraw7.0.6/JHotDraw7.0.6/src/org/jhotdraw/samples/draw/DrawLiveConnectApplet.java" startline="45" endline="134""
pcid="619"></source>
<source file="C:/Systems/JHotDraw7.0.6/JHotDraw7.0.6/src/org/jhotdraw/samples/svg/SVGLiveConnectApplet.java" startline="44" endline="131"
pcid="364"></source>
</class>
```

Figure 3.3: A Sample NiCad Clone File

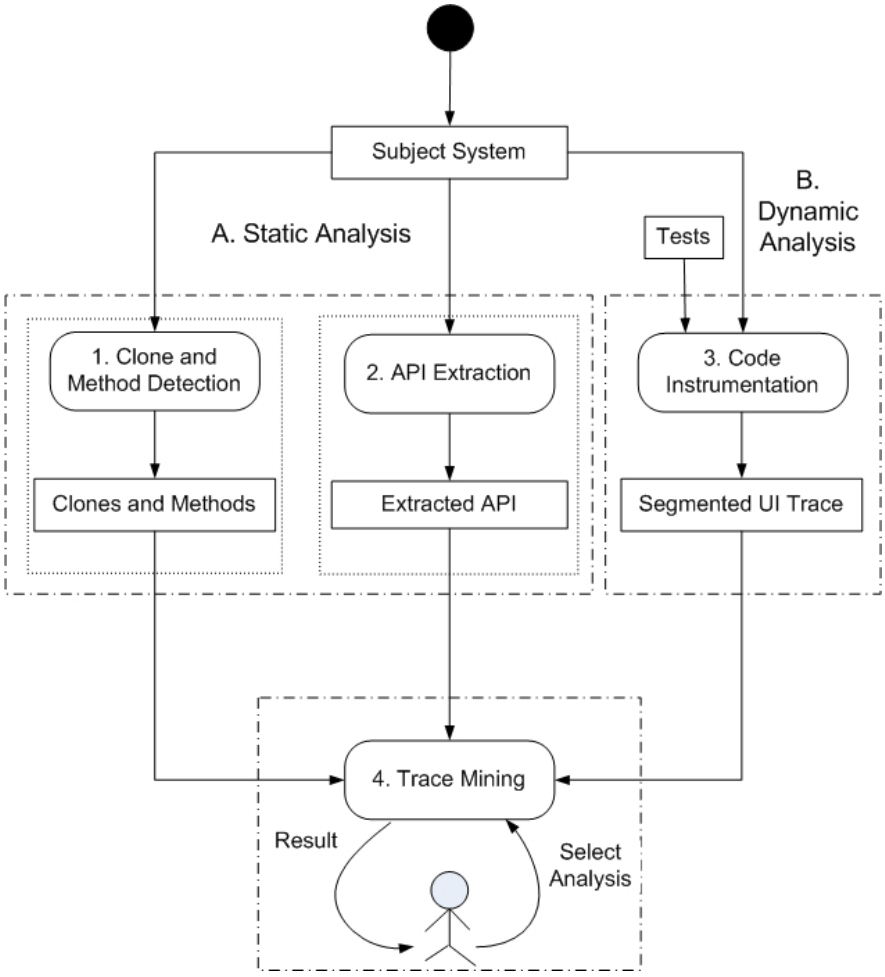

Figure 3.4: Analysis Framework

implement certain functionalities provided by the IDE itself. The import lists are further processed to get the class names from them which are further processed to extract the methods in them.

3.4.2 Dynamic Analysis

Dynamic analysis [14] is the technique of analysing the data collected from a running program. The advantage of using dynamic analysis is that it exposes the system's actual behaviour. However, the drawback of it is that only partial picture of a system is revealed and the obtained information are dependent on the inputs used to exercise the system's functionalities. Dynamic analysis is used in a variety of applications such as software testing, performance analysis, and program comprehension through reverse engineering. We are interested in collecting a system's behavioural information in the form of execution traces. This can be done by instrumenting the code, monitoring the execution environment, or as is the case in our approach, by dynamically weaving aspects to log runtime information. A challenge is to have an adequate set of tests (use sets) or recorded user activities to collect relevant behavioural information.

The first part of our dynamic analysis approach is to instrument the code for collecting the execution traces. We use aspect-oriented programming [42] and dynamic loading to log method calls. This allows us to generate a sequence of execution traces that will later enable us to identify the component interactions, call graphs and metrics using trace mining. To capture the execution traces, the system is run using a set of tests corresponding to the activity we wish to analyse. By focusing on a single maintenance task, the test corresponds directly to the user interaction that is being investigated.

As part of the code instrumentation step of the dynamic analysis phase, we use an AspectJ [42] implementation to describe and capture the interface actions that occur in a target application. We have written a tracing aspect that weaves with the program and records when any method in the target application is invoked. The advantage of using an AspectJ implementation is that it does not turn off the just-in-time compiler, and enables access to any component (method, parameter, constructor, etc.) in a software system.

To aid in program comprehension, we also segmented the execution trace into meaningful segments called user interface (UI) traces [89]. In this regard, we have categorized traces into two levels of segments called test segment and event segment. An event segment contains UI traces that belong to events such mouse events, mouse clicks, keyboard events, display updates events, thread signal events and launch events. A mouse event is an event by which we mean any sort of mouse actions such as movement of mouse cursor (mouseEntered or mouseExited) over any part of a component. However, a mouse click event signifies events associated with clicks on any button in the user interface. Similarly, a thread signal corresponds to any new threads spawned to carry out a particular functionality while a display update event fires when changes to display in the screen is done.

On the other hand, a test segment contains traces corresponding to a particular test from a use set to run the system. A test segment contains a set of events and each event contains a set of methods. To achieve this process we have developed a technique called annotation as shown in Figure 3.5. It allows us to mark

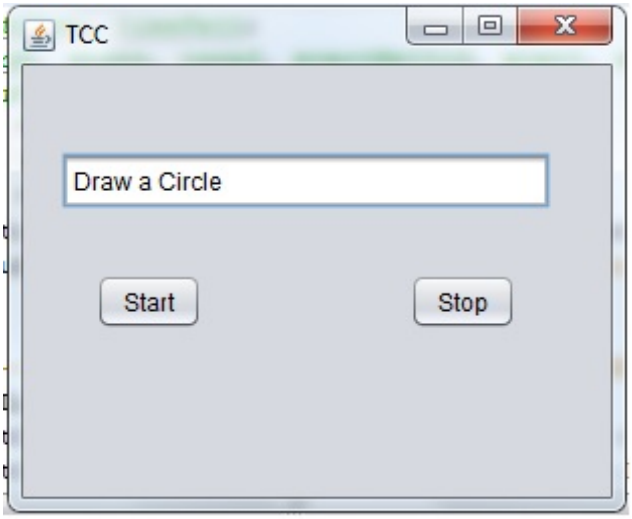

Figure 3.5: An Interface of Trace Annotation

the start and end of a test a user is performing during the runtime of the system. Figure 3.6 shows a sample output trace which is collected during the instrumentation process. As we can see the trace file contains sequences of method calls segmented under the tag "Start of the Activity" and "End of the Activity" for a particular test. The tagged block is further segmented for the associated events with tag "Start of the Event" and "End of the Event". Each trace in the block is accompanied by a hashcode, a unique identifier for the trace, arrow to denote the call graph of ($--> $ means start of execution, $<-- $ means end of execution). The method calls are represented by the type of method, their absolute path, the type and the parameters lists. Therefore, traces are enclosed under the event blocks which are enclosed under the test blocks.

3.4.3 Trace Mining

This is the final phase of our framework which involves the end-users. It is triggered by user's selection on what they want to know about a system's characteristic in order to understand the implementation. The run time data and the static data are together used to help users understand the system's implementation, their interactions and the dynamics on how they behave using a set of different types of metrics.

3.5 Trace Mining Framework

To explore the nature of the traces and to reveal how a system is implemented we have divided the mining process into three sub-processes as shown in Figure 3.7. They are as follows: 1) Method and clone mapping 2) Metrics calculation, and 3) Analysis selection.

```
Start of the Activity-Launch
Start of the Event- Launch*1290300832
-->#1290300832:void C:/Systems/JHotDraw7.0.6/JHotDraw7.0.6/src/org/jhotdraw/samples/draw/Main/main(String[]):1
 -->#49272146:C:/Systems/JHotDraw7.0.6/JHotDraw7.0.6/src/org/jhotdraw/beans/AbstractBean():1
 -->#1335753727:java/beans/PropertyChangeSupport(Object):1
 <--#1335753727:java/beans/PropertyChangeSupport(Object):1
 <--#49272146:C:/Systems/JHotDraw7.0.6/JHotDraw7.0.6/src/org/jhotdraw/beans/AbstractBean():1
 -->#218843951:C:/Systems/JHotDraw7.0.6/JHotDraw7.0.6/src/org/jhotdraw/app/AbstractApplication():1
 -->#1618280793:java/util/LinkedList():1
 <--#1618280793:java/util/LinkedList():1
 -->#227180285:java/util/LinkedList():1
 <--#227180285:java/util/LinkedList():1
End of the Event- Launch*1290300832
Start of the Event- mouseEvents (JPopupButton)  -14*606144987
-->#606144987:void C:/Systems/JHotDraw7.0.6/JHotDraw7.0.6/src/org/jhotdraw/draw/action/JPopupButton/FormListener/mouseEntered(MouseEvent):14
 -->#606144987:void C:/Systems/JHotDraw7.0.6/JHotDraw7.0.6/src/org/jhotdraw/draw/action/JPopupButton/FormListener/mouseEntered(MouseEvent):14
 <--#606144987:void C:/Systems/JHotDraw7.0.6/JHotDraw7.0.6/src/org/jhotdraw/draw/action/JPopupButton/FormListener/mouseEntered(MouseEvent):14
<--#606144987:void C:/Systems/JHotDraw7.0.6/JHotDraw7.0.6/src/org/jhotdraw/draw/action/JPopupButton/FormListener/mouseEntered(MouseEvent):14
End of the Event- mouseEvents (JPopupButton)  -14*606144987
End of the Activity-Launch
```

Figure 3.6: Diagram of a Sample Trace

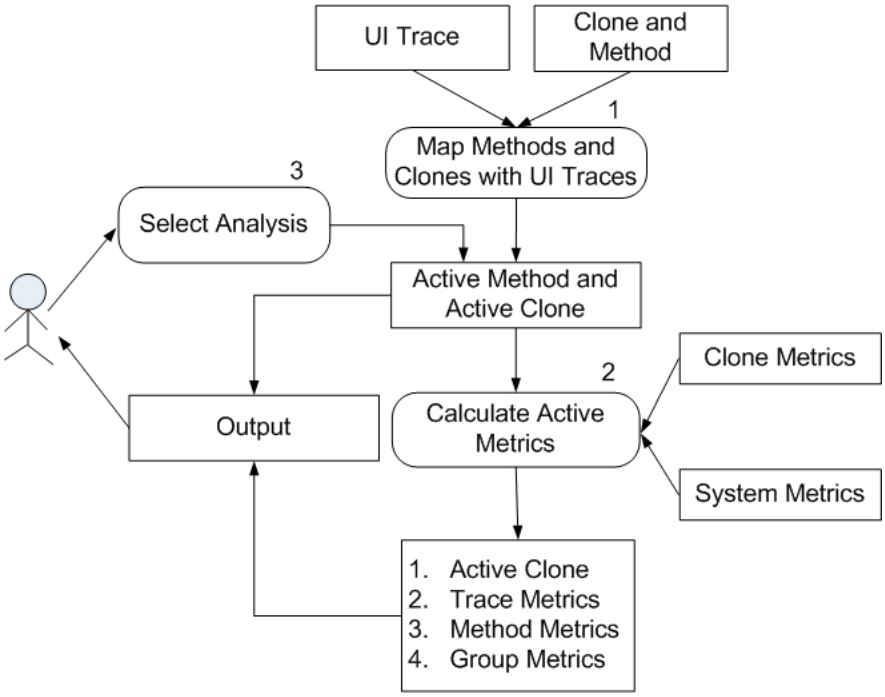

Figure 3.7: Trace Mining Framework

3.5.1 Clone and Method Mapping

This step marks the initial process of the trace mining. It maps the trace information with the source code. In case of clone mapping for each method in the trace, if it contains a clone fragment in the source code, the corresponding clone class/pair is marked as 'active'. This step determines which clone classes are active corresponding to a test. First, using dynamic analysis we record the traces of the system put into execution using each individual test. Each trace record in the trace files consists of the full class name, class type, method name, and parameter types. Second, we get all the clones of the subject system returned by NiCad [77] clone detector. Third, we determine whether the trace related code contains any clone fragments. For clone mapping, we created a script to extract all the method names from the NiCad XML file. The clone fragments for which we get a match in the trace are called active clone fragments, and then we retrieve the corresponding clone classes to which the active clone fragments belong to. These clone classes are treated as active clone classes. Figure 3.3 shows a sample file generated by the clone detector in the XML format. A set of of similar fragments are grouped together into a class with a unique identifier called class id. Each fragment is also associated with starting and ending line numbers. The approach of method mapping is similar to our approach of clone mapping. This stage determines which of the internal and external methods are active. That is, we consider all the extracted methods and their enclosing classes to mark them as active if they are found in the trace. This applies to both the external and the internal methods. The methods that are found in the trace are known as active methods while the set of packages, classes and methods together is termed as active API. Once the active methods are marked the mapping also yields the code block associated with the internal methods.

As part of the mining process we have clustered the method calls in the traces into two types: External methods and Internal methods. Internal methods are the ones that are developed by the developers with a user defined method names with or without parameters and return values that belong to a user defined packages and classes for the purpose of implementing a certain functionality. On the other hand, external methods are those methods that are provided by the development environment such as Netbeans, Eclipse and so on with the same representation as the internal methods with parameters, and return values. They are accessed through the declaration of packages and classes at the import list. The main difference with the internal methods is that in case of external ones the source code is not available. In our case we term external methods to those methods that have a prefix of 'java' or 'javax'. For instance, a method with the name 'readline()' belonging to the class 'BufferedReader()' and the package 'java.io' is considered as an external method because it can be expressed as java.io.BufferedReader.readLine() which has a prefix of 'java'. Once the methods are clustered they are further processed and categorized to tag them into groups. The external methods are categorised into groups by application programming interface and the internal methods into active clones.

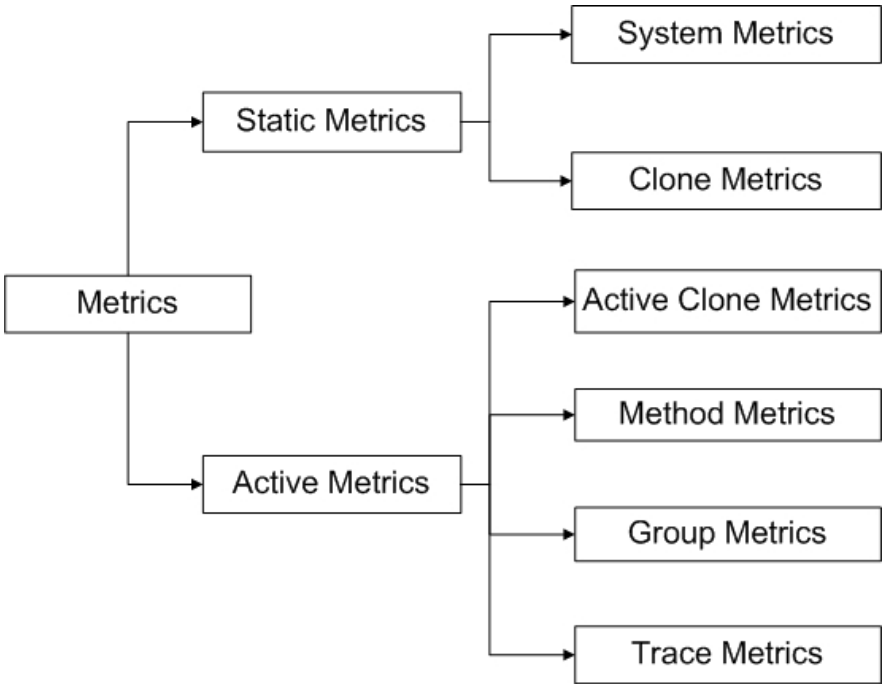

Figure 3.8: A Taxonomy of Metrics Classification

3.5.2 Metrics Calculation

We use metrics to quantify aspects of both static and dynamic analyses. Some of the metrics we have proposed in this thesis and some are adapted from relevant empirical studies [26]. Metrics are defined for both method and block clones. To assist in the mining process we have classified metrics into static and dynamic metrics. Static metrics are composed of system and clone metrics while active metrics is a collection of active clone metrics, method metrics, group metrics and trace metrics. Static metrics are those metrics that calculates the components at the source code level. Active metrics on the contrary is dependent on the execution traces. It calculates the behaviour of the collected components as traces after a system is executed. The taxonomy of the metrics are, therefore, depicted in Figure 3.8.

System Metrics

- L_{TOTAL}: total number of lines of code in a system.
- F_{TOTAL}: total number of files in a system.
- M_{TOTAL}: total number of methods in a system.
- B_{TOTAL}: total number of blocks in a system (minimum of 5 LOC).

Clone Metrics

- N_{CLONE}: number of clones in a system.

- CC: number of clone classes in a system.
- $\%M_{CLONE}$: percent of methods that contain a clone.
  $\%M_{CLONE}= N_{CLONE}/M_{TOTAL}$
- $\%B_{CLONE}$: percent of blocks that contain a clone.
  $\%B_{CLONE} = N_{CLONE}/B_{TOTAL}$
- F_{CLONE}: number of files that contain a clone.
- $\%F_{CLONE}$: percent of files that contain a clone.
  $\%F_{CLONE} = F_{CLONE}/F_{TOTAL}$
- L_{CLONE}: lines of code that are part of a clone.
- $\%L_{CLONE}$: percent of lines that are part of a clone.
  $\%L_{CLONE} = L_{CLONE}/L_{TOTAL}$
- B_{CC}: number of tests into which a clone class is active.

Group Metrics

- CPU_{GROUP}: total CPU time consumed by methods in a particular group.
- N_m: number of active methods in a group.
- E_{GROUP}: total number of executions of active methods in a group.
- Br_{GROUP}: number of tests into which a group is active.

Trace Metrics

- N_p: total number of files invoked in a trace. When a method from a package is invoked the package is considered to be invoked.
- N_c: total number of classes invoked in a trace. When a method from a class is invoked the class is considered to be invoked.
- N_{int}: total number of distinct internal methods invoked in a trace.
- N_{ext}: total number of distinct external methods invoked in a trace.
- $\%F_{int}$: percent of packages containing internal methods invoked in a trace.
  $\%F_{int} = N_p/F_{TOTAL}$

- $\%F_{ext}$:percent of classes containing external methods invoked in a trace.
  $\%F_{ext} = N_c/F_{TOTAL}$

- CPU_X: total CPU time consumed by X. Here X corresponds to system use, test and event.

- $\%CPU_X$: percent of CPU time consumed by X.
  $\%CPU_X$: $= CPU_X/CPU_{TOTAL}$

Method Metrics

- CPU_{METHOD}: total CPU time consumed by a particular method in a trace. The method can belong to any event, tests and system use.

- E_{TOTAL}: total number of times a particular method executed over all the tests.

- $\%CPU_{METHOD}$: percent of CPU time consumed by a particular method during the system runtime.
  $\%CPU_{METHOD} = CPU_{METHOD}/CPU_{TOTAL}$

- Br_{METHOD} = total number of tests in which a particular method is invoked.

- L_{METHOD} = total number of lines of code for a particular invoked internal method.

Active Clone Metrics

- N_{ACLONE}: total number of active clone fragments.

- CC_{ACLONE}: total number of clone classes that contain at least one active clone fragment.

- $\%M_{ACLONE}$: percent of invoked methods that contain at least one active clone fragment.
  $\%M_{ACLONE}$= N_{ACLONE}/N_{int}

- F_{ACLONE}: number of files that contain at least one active clone fragment.
  $\%F_{ACLONE} = F_{ACLONE}/F_{TOTAL} * 100$

- L_{ACLONE}: lines that are part of an active clone fragment.
  $\%L_{ACLONE} = L_{ACLONE}/L_{TOTAL} * 100$

3.5.3 Analysis Selection

This phase is one of the important phases that triggers the trace mining and is controlled by the end users. Once a user selects the type of analysis the mining occurs accordingly. The selection of analysis thus corresponds to one of the features in the TrAM being activated by the user. The features include choices based on selecting, filtering, searching, grouping, interface snapshots and corresponding code viewing. The users are also able to choose and analyse information of the trace on different levels of granularity such

as tests, events and methods. Once an analysis is selected the corresponding information of test, events, methods, snapshots and metrics are displayed as an output to the user.

3.6 Test Design

The main objective of testing in maintenance is to assess the quality of a system. To do so, a maintenance engineer must have a set of test suites. A test suite thus will contain a number of test cases and each test case will correspond to what features of the system are tested. Assessing the quality of a system means finding how its functionalities confer with the customer requirements and how reliable it is in terms of bugs. To achieve this a good set of test suites with broader coverage is desired. The more methods are executed the more features can be located and better comprehension can be achieved. Our objective here is not to find bugs or asses the quality of a system, but to exercise as much methods in the system in order to understand the implementation of the features. In our study we have used both synthetic and actual test suites. We have studied all the features of a system from the documentation provided. For instance, in case of JHotDraw we have collected all the features associated with a system use drawing. That is, what are the possible objects for the system use 'draw a object' such as circle, rectangle, polygon and so on which corresponds to tests. Each test is used as an input to the system and the traces are recorded for analysis.

3.7 Trace Mining Data Organization

3.7.1 Model Schema

In order to organize our trace mining data we have represented them as an entity relationship diagram. A total of eight entities are required in order to organize the whole set of information. The entities are subject system, system use, test, event, method invoked, group, group element, and method. An entity is a table in the database and the attributes qualify an entity. The entities are related to each other using a relationship. A relationship can be of four types: one-to-one, one-to-many, many-to-one and many-to-many. In our model, the relationships are such that every subject system will contain a number of system uses and therefore a one-to-many relationship exists between them. Every system use may again contain a number of events making them one-to-many as well and every event will contain more than one invoked methods and thus creating a one-to-many relationship between them as well. Every subject system will also contain more than one groups making that a one-to-many and every subject system is also comprised of multiple methods and a one-to-many relationship between them too. The entity method and group element also behave as one-to-many as a single method can also belong to multiple methods. The group and group element are tied together with a one-to-many relationship since a single group will hold more than one methods. Similarly a single method in the method entity may be invoked in a series of different events, therefore a one to many relationship also exists between them. Each entity contains a number of attributes as follows:

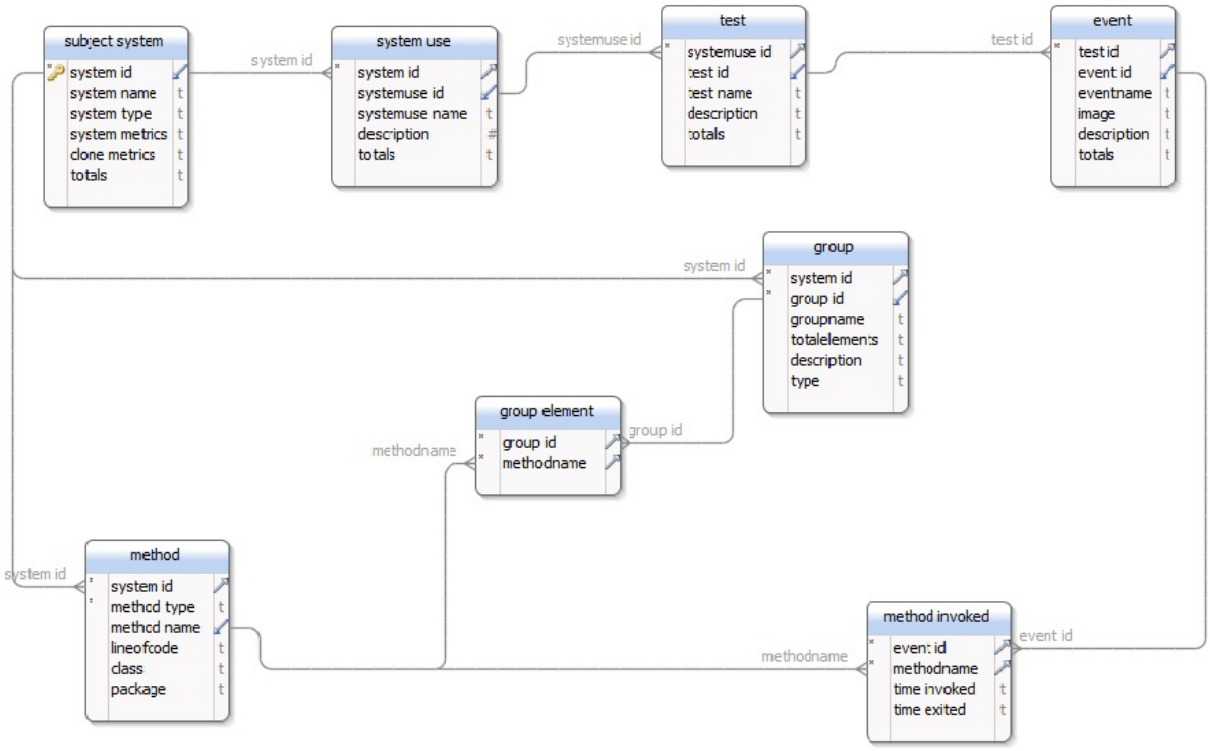

Figure 3.9: An Entity Relationship Diagram of the Data Organization of TrAM

a. **subject system** : $\langle systemid, systemname, systemtype, systemmetrics, clonemetrics, totals \rangle$

b. **system use** = $\langle systemid, systemuseid, systemusename, descriptions, totals \rangle$

c. **test** = $\langle systemuseid, testid, testname, description, totals \rangle$

d. **event** = $\langle testid, eventid, eventname, imagepath, description, totals \rangle$

e. **method invoked** = $\langle eventid, methodname, totaltime \rangle$

f. **group** = $\langle systemid, groupid, groupname, totalelements, description, type \rangle$

g. **group element** = $\langle groupid, methodname \rangle$

h. **method** = $\langle systemid, methodtype, methodname, lineofcode, class, package, methodblock \rangle$

Entity: subject system

This entity holds the information of all the detailing of a system such as JHotDraw, HSQLDB and so on.

a. system id: identifies each of the subject system with an id (1, 2, 3)

b. system name: holds the name of the system (JHotDraw, HSQLDB, argoUML)

c. system type: holds the type (game, graphics, database, editor)

d. system metrics: the metrics that qualifies the systems such as total lines of code, total number of methods, total number of files, total number of blocks, total imported packages.

e. clone metrics: metrics that qualifies the similar code fragments such as number of clones, clone class, clone line of code and so on.

f. totals: stores the trace metrics as follows:

. totalsystemuse: holds the value for the total number of test suites used in the system for testing

. totalinvoked_internalmethods: total number of internal methods invoked in the trace

. totalinvoked_externalmethods: total number of external methods invoked in the trace

. totalinvoked_internalclass: total number of internal class invoked in the trace

. totalinvoked_externalclass: total number of external class invoked in the trace

. totalcputime: holds the total number of cpu time consumed by all of the invoked methods

. totallineofcode: holds the total lines of code active in the system

Entity: system use

This entity carried the information of all test suites used in the system during program execution.

a. system id: again this signifies the system identification

b. systemuse id: unique identifier for each use of the system

c. systemuse name: holds the name of the system use (systm use1, system use2)

d. description: stores any further comments about each record in the table.

e. totals: stores the trace metrics and is the same as totals in the entity subject system.

Entity: test

Stores the information about the tests corresponding to each test suite

a. systemuse id: uniquely identifies the system use

b. test id: uniquely identifies the tests

c. test name: holds the name of the tests (draw a circle, change line color)

d. description: stores any further comments about each record in the table.

e. totals: stores the trace metrics.

Entity: event

Stores the information of every user event activated during the execution.

a. test id: uniquely identifies the tests

b. event id: uniquely identifies the events within each test case. Each event id is generated automatically during program execution. Same events will have different event id when activated at some other point of time during the execution. For instance a thread signal may be in two different test cases such as draw a circle and change fill colour. Though they are exactly the same in behaviour they are identified as a different event with different event id.

c. eventname: holds the name of the events (launch, thread signal, mouse click)

d. image: the absolute path of the corresponding image. Each image is identified using the event id.

e. description: any comments that would describe the record.

e. totals: stores the trace metrics.

Entity: group

This entity stores the information of the groups of methods. Each internal and external contains a number of groups.

a. group id: uniquely identifies each group

b. groupname: names of the groups of internal and external methods. For instance internal methods may have groups of string, io, swing and so on while internal methods may have groups of classes such colorIcon, PalletteMenuItem and so on.

c. totalelements: holds the information of the total number of elements contained in each group.

d. description: any comments that would describe the record.

e. type: holds the information about the type of group such as if the group belongs to a clone class one then type will be Type 1, Type 2, Type 3 and if any other group the type will be Type 0.

Entity: method invoked

This entity organizes data of the methods invoked in the trace.

a. event id: unique identifier for the events

b. methodname: holds the names of the methods invoked in the trace

c. time invoked: time when a certain method was invoked

d. time exited: time when a certain method finished execution

Entity: group element

This entity further qualifies the group table

a. group id: uniquely identifies the groups

b. methodname: names of methods contained in the group

Entity: method

Stores data of groups that qualifies it in terms of clone types

a. system id:

c. method type: indicates the type of methods (internal or external)

d. lineofcode: holds the number of lines of code for each method

e. class: it identifies to which class the methods belong to

f. package: it identifies the packages (for instance packages found at import list for external methods, and file names to which a method belong to signifies package for internal methods)

Chapter 4

Trace Analysis and Management-from Design to Implementation

4.1 Introduction

In this chapter, we would present our tool, TrAM (Trace Analysis and Management). Our tool presents the information about a systems implementation and its dynamics to the user easily in a meaningful and concise way. The initial phase that comprises the trace mining, processes all the data and makes them available to provide insight of the system at the implementation level, a challenge often encountered by the developers during the maintenance activity. Therefore, the presentation of information marks the second phase of our tool. After the trace mining phase, the mined information can be viewed and analysed using different navigation and filtering techniques. The visualization phase not only supports the users in understanding how a system is being implemented and how to identify a specific feature in it but it also provides ways to observe any part of the program execution at various levels of detail such as tests, events and methods. One of the major problems with a large volume of execution traces is that the analysis is cumbersome and difficult. Often the users get overwhelmed and out of focus with the trace data in hand. TrAM thus permits modular yet detailed view of program implementation using selective controls in its interface.

4.2 Visualization Architecture

The prototype incorporates various features such as filtering, searching, grouping (i.e., categorization), image and code viewing, and metrics viewing that allow users to investigate the behaviour of methods (external and internal). How useful a tool is depends on what features does it provide, how the information is presented to the users, and how accessible the tool is in terms of its use. Based on these three criteria we have classified the architecture of our tool as follows: feature-specific architecture and presentation-specific architecture.

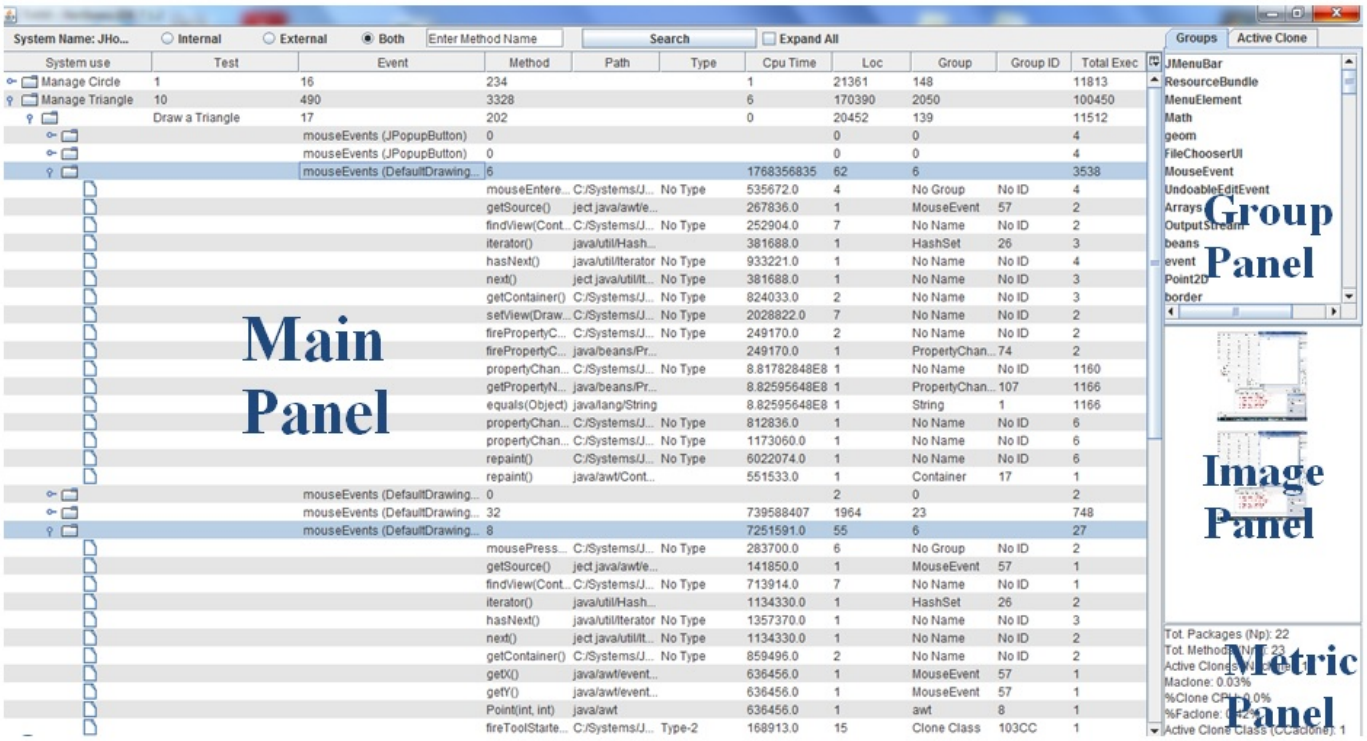

Figure 4.1: Diagram of the Trace Panel of TrAM

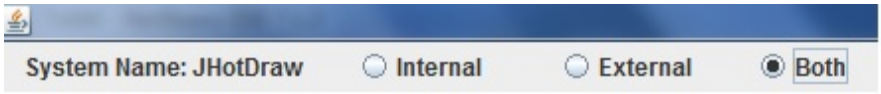

Figure 4.2: Diagram of the Filtering Feature of TrAM

4.3 Feature-specific Architecture

Feature specific architecture signifies what features are available in the interface and what are their functionalities. Manipulating traces is a challenging task due to the sheer volume of data which in turn makes the mining task difficult. Thus trace management is one of the key factors that impacts program comprehension. Traces are very rich in information and the basis for an immense data warehouse. If mined they can help us understand a system's runtime behaviour. Besides the challenge of managing traces, a good way of presenting the information and techniques of making them easily accessible to the users need to be available. Therefore, in our tool we have implemented the following features to facilitate the users in navigating through traces: filtering, searching and categorizing.

4.3.1 Filtering

As part of trace management, we have sub-divided the method calls in traces into two categories: External methods and Internal methods. Internal methods are the ones that are developed by the developers with

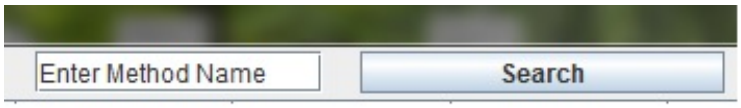

Figure 4.3: Dialog Box for the Searching Feature of TrAM

an user defined method names with or without parameters and return values that belong to a user defined package and class for the purpose of implementing a certain functionality. On the other hand external methods are those methods that are provided by the development environment such as Netbeans, Eclipse and so on with the same representation as the internal methods with parameters, and return values. They are accessed through the declaration of packages and classes through imports during program development. The main difference with the internal methods is that in case of external ones the source code is not available. In our case we term external methods to those methods that have a prefix of 'java' or 'javax'.

To implement the feature as in Figure 4.2 the distinctions between types of methods is done using radio buttons. The buttons allow the users to filter the trace based on the selections. As a result, filtration can be achieved in three different ways such as internal, external and both. On selecting the internal button only traces with internal methods are displayed and the external button shows only the external methods in the trace. However, both buttons display the complete trace containing the external and the internal methods.

4.3.2 Searching

One of the main challenges in dynamic analysis is the sheer volume of traces that are collected during the run time. To understand a system's implementation, a user also needs the location of the implemented methods i.e., to which event and test the method was invoked. A method may be invoked a number of times in the same event of the system use (tests in our case) or in different events in different system uses. Therefore, instead of looking for other methods a user may try to know about a particular method and the events where it is invoked. To facilitate this, a search feature has been added containing a text field and a button. The text field allows the user to enter any method name they wish to locate in the trace. Once the search is executed the entire trace is parsed and the methods appear in the display for analysis. The search option as shown in Figure 4.3 is to complement the feature analysis with more precise location of the methods according to user requirement.

4.3.3 Categorization

To facilitate more in the comprehension process, our tool incorporates the categorization feature as shown in Figure 4.4. During a system's development, developers use the application programming interfaces that are packaged by the development environment. It is therefore interesting to know about the APIs that are being used in the development and at the same time what APIs have been invoked during the run time while executing a particular use. Group wise information about the API would inform the developers about the

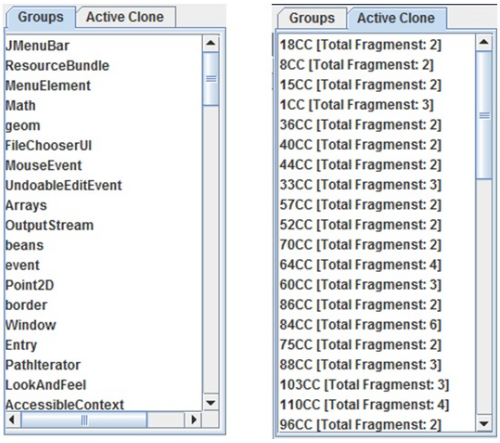

Figure 4.4: Diagram of the Group Feature of TrAM

most popular APIs being invoked. It will also enable them to find out the context wise usage of APIs in a visual format. The categorization is achieved through grouping to categorize the method calls by application programming interface, and to group internal method calls as being active clones or not. For example, a method call to a math library (e.g., min(...), abs(...), or round(...)) would be automatically tagged as an external method call and part of group Math. If a call to an internal method is a clone, it is tagged as an active clone and tagged with its active clone class. API and clone categorization are done automatically using static analysis techniques. Groups allow us to filter traces and metrics by group or clone class, by simply selecting the groups or classes of interest in the group/clone panel. Figure 4.4 shows the categorization of external methods and the internal methods. They are represented using two different tabbed panes, 'Groups' and 'Active Clone'. The 'Groups' tabbed pane shows the list of groups of external methods while the 'Active Clone' tabbed pane represents the active clone classes and the total fragments contained in it.

4.4 Presentation-specific Architecture

Presentation-specific architecture signifies the organization of the tool. The usability of a tool depends on how easily a user can use the system and at the same time how well the information is organized. The more presentable and informative the visuals are the more helpful will it be for the users to interpret the result and understand a system's implementation. Based on the design implication the tool includes the following four panels as shown in Figure 4.1:

F1 *Main panel*

F2 *Image thumbnail panel*

F3 *Total metrics panel*

F4 *Groups panel*

4.4.1 Main Panel

The main panel is the largest area which includes the display of the mined traces collected during the runtime. The traces are organized into a hierarchical tree-like structure. The hierarchy is composed of four different levels including system use, test, events and methods. Each system use is composed of multiple test cases, each test case includes multiple events and each event contains a number of invoked methods. The invoked methods contain both the internal and the external methods which can be further filtered out using the type filtration buttons. The panel is designed over the idea of tree table where each row represents the hierarchy of system use, tests, events and methods. The columns hold the values that qualify the invoked methods further.

The y-axis of the panel is occupied by a series of columns that hold values for each individual levels of traces in the hierarchy. The columns are *systemuse*, *test*, *event*, *method*, *path*, *type*, *cputime*, *loc*, *group*, *groupid*, and *totalexec*. The *systemuse* holds the information of all the system use of a particular system such as 'manage a triangle' which includes tests in the *test* column such as 'Draw a triangle', 'Save file' and so on. *Events* constitute the information of the actions that are performed such as to 'Draw a triangle' or 'Save file' and *methods* contain the series of method calls within the *event*. These sequences of method calls together are termed as call graph which also show the caller-callee relationship. The column *path* shows class to which each method belongs to. For internal methods the path gives the information of the absolute path name of the file to which the methods belong and in case of external methods the path is the whole package name of the method. For instance, let us consider an internal method 'getContainer()' that has a path information of C:\Jhotdraw7.2.6\src\org\jhotdraw\draw\DefaultDrawingView. It means that the method 'getContainer' belongs to the class 'DefaultDrawingView'. Similarly an external method 'iterator' having a path information of java\util\HashSet means that the method 'iterator' belongs to the class 'HashSet'. The column 'type' further qualifies the invoked method with the information of whether it belongs to any type or not. The type information is applicable to internal methods only and more specifically to those internal methods that are part of a clone class. The clone detector NiCad organizes the clone data as a group of methods within a clone class. Each clone class therefore corresponds to a type (e.g. type 1, type 2, and type 3). In the main panel a method belonging to clone class shows the type information while a method that does not belong to any clone class has the type value of 'No Type'. In case of external methods the type remains blank. The runtime data such as time spent during execution of method and number of times a method is invoked is displayed by the columns *cputime* and *totalexec*. column. On the other hand in order to further enhance the property information of a method, each method is again tagged with a *group* and *groupid* information. However, the *group* is applicable fully for the external methods and partially to the internal methods. In case of internal methods if a method belongs to a active clone class then its group value will be 'clone class' otherwise it will be 'No Name'. External methods will have the values that will reveal to which class they belong to such as 'String', 'Util', IO' and so on. Each *group* also contains a *groupid* and for the external methods the *groupid* will be any unique arbitrary value but for the internal ones the *groupid*

will be the active clone class id which they belong to. If none of the methods belong to any active clone class then the id information will be 'No ID'.

Panel Operation

As described in the previous section the main panel offers a wide two dimensional view in the screen with a set of values displayed in the columns the quantifies the traces in four different levels of granularities (system use, test, event and method). Besides providing a clear wide screen for the users to see and analyse data, the main panel also supports various operations allowing users to operate from. The main motivation behind the wide screen for the main panel is to provide a complete view of the trace and at the same time offer a simple interface to allow the users to navigate through the various levels of granularities as chosen by them. To achieve this we have incorporated the operations in the main panel which are follows:

F1 *Selecting*

F2 *Collapsing*

F3 *Column manipulation*

Selecting

This is the primary and the most important option that provides flexibility to the end users to choose their desired level of traces in order to carry out their analysis. It is the starting point of the mining that initiates the analysis process. To facilitate the navigation process the traces are managed in a tree-like structure. First, once the traces are loaded the users get a concise view of the topmost level (i.e., system use). The tree structure containing a root handle allows further navigation into the deep level of the hierarchy by just clicking on it. The root handle facilitates both the opening and the closing of the tree at the desired level. At some point in the hierarchy the handle disappears indicating the last level containing the invoked methods.

Collapsing

One of the main challenges in dealing with traces is its immense volume. Even a smallest segment of code when executed produces immense amount of lines of text that makes difficult to manage and comprehend. Once the traces are loaded in the main panel each node of a tree represents a system use. At some point in the evolution of software more features may be added and thus more test. This will ultimately pose threat to the tree-like structure at the main panel making it more intense and occupying a greater portion of the screen as well. Moreover, at the same time a user may have to navigate though the different levels of structural hierarchies by opening a large number of tree nodes. Thus walking through the tree nodes may make the screen too much crowded with information. In such case a user may try to restart the navigation. This can be achieved by closing all the nodes individually which also is a cumbersome task. To ease the process we have provided a feature to collapse and expand the tree using a checkbox. Once the checkbox is selected

all the tree nodes get expanded and when it is deselected the nodes get collapsed in the screed space. Once all the nodes are collapsed the user then can start over the navigation by choosing their desired nodes for analysis.

Column Manipulation

The main panel presents a series of columns with information that quantifies the traces which are packed together. Therefore, the value of one column may get overlapped by the other column hiding the information. To facilitate the viewing of the values of all the columns the design provides a way for column manipulation. The users can select the columns of their choice to keep in the screen and remove the rest from the view. This way the screen allows the flexibility to close the columns that are not required for the task at hand during the analysis.

4.4.2 Image Thumbnail

Feature identification is one of the important requirements in program comprehension. The first thing a user will want to know about a system's implementation is the portion of code that corresponds to a particular functionality. To provide a way for clear understanding of the portion of code that executed an image thumbnail panel is created. The panel can accommodate more than one thumbnails all ordered sequentially. The panel is populated with the thumbnail views based on selection of events in the main panel. The order of thumbnails corresponds to the order of events in the main panel. The main motivation behind keeping the thumbnail panel is to allow the users to have a glimpse of the images and provide them with the flexibility to start analysis by clicking on any one of the thumbnails.

4.4.3 Image and Code Viewer

Once a thumbnail is selected from the thumbnail viewer by double clicking, a new big screen called the image viewer is opened containing the detailed view of the thumbnail. From the Figure 4.5 we can see that the image viewer displays only one image at a time and it might be interesting for the user to navigate through all of the thumbnails in the thumbnail panel and at the same time view the enlarged image in the image viewer. To synchronise the process the image viewer offers two sets of buttons called the 'Back' and the 'Next' button. These two buttons will allow the users to navigate through the selected thumbnails by moving back and forth as desired by them. As a result they will be able to have clear view of the images in details and at the same time can compare each image from one another. Once the user switches to the image viewer screen they also need to recall to which event, test and system use the images belong to. One way to do so could be to switch back to main panel and see the selected events. To ease the switching back and forth between the main panel and the image panel, the image panel displays the whole path of the hierarchy in the X− >Y− >Z format at the top of the screen in the form of caption. Here X, Y and Z represents system use, test case and event respectively. For instance Manage Triangle− >Draw Triangle− >MouseEvent.

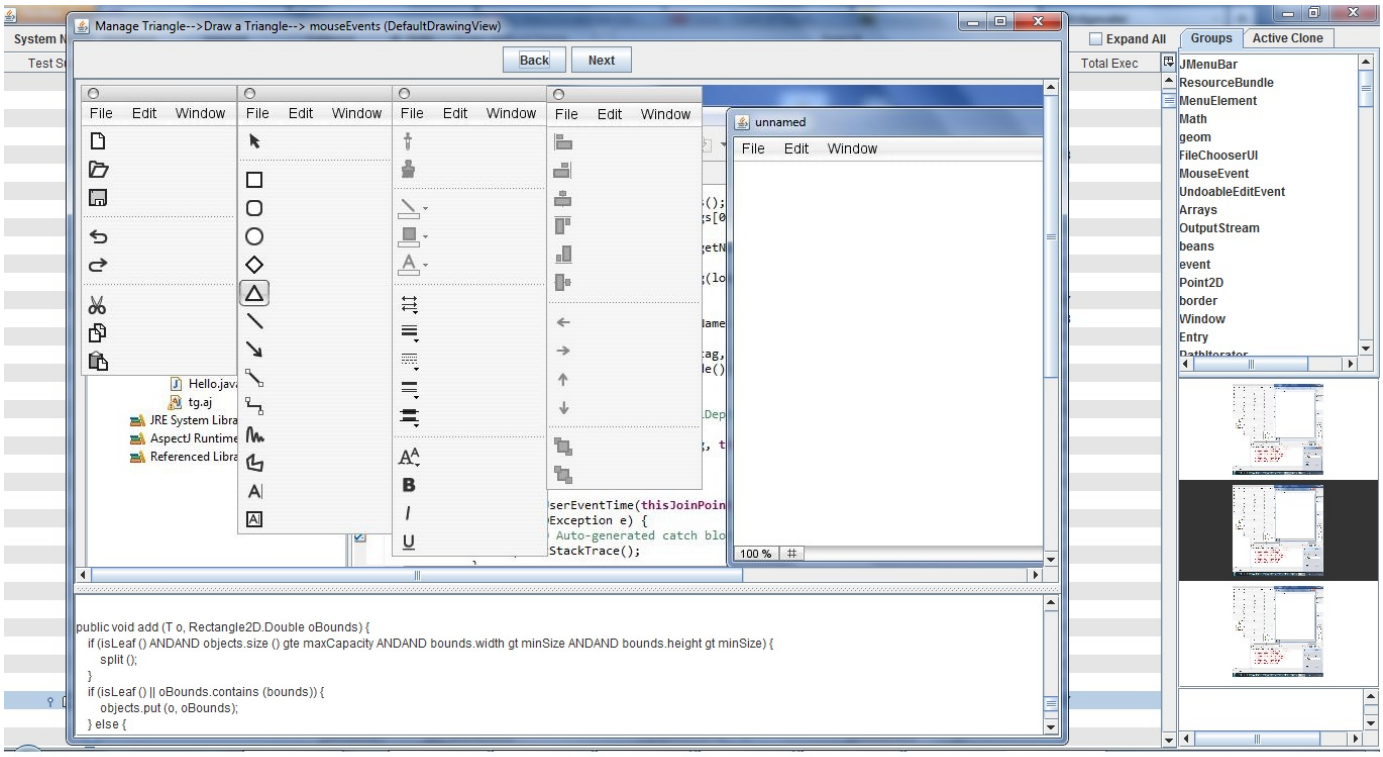

Figure 4.5: Diagram of the Image and Code Viewer

After knowing the trace and the images associated with the events one would certainly like to know about how the user interface corresponding to the feature is implemented. Although they already get to know about this using the methods invoked but the actual code written by a developer and in what way the methods are implemented would be the last thing that they would put emphasis on towards understanding the system.

The main panel gives the call graphs of the methods invoked contained within the events. These call graphs correspond to executed methods that may or may not include every path of execution during the runtime of the system. For example, a method block contains an if-else block, but during the runtime only one of the paths will be traversed at a time. Therefore, to assist in understanding the portions of the code, our tool incorporates a code viewer in conjunction with the image viewer. The code viewer which is attached to the image viewer displays the block of code corresponding to the methods invoked in the call graph. The code contained in it is the concatenation of all the method blocks corresponding to the methods invoked in a particular event. The code viewer is dependent on the image viewer and is updated when a user navigates back and forth through different images loaded in the image viewer using the 'Back' and 'Next' button.

4.4.4 Metric Viewer

The metric viewer panel (Figure 4.7) provides space for displaying properties of traces using a set of metrics. Program comprehension can be further enhanced by the use of metric values on the system. One of the

```
gettingURLResponse(String url) {
  URL serverUrl = new URL(url);
  WebConversation conversation;
  conversation = new WebConversation();
  WebRequest request;
  request = new GetMethodWebRequest(serverUrl,"");
  return conversation.getResponse(request);
}
```

a

```
gettingURLResponse(String url){
  return new WebConversation().getResponse(url);
}
```

b

Figure 4.6: Comparison of Two Different Coding Style (taken from [41]) (a) Code Snippets of an Inefficient Coding Style (b) Code Snippets of an Efficient Coding Style

main notions towards using dynamic analysis is also to profile the behaviour of the system by mining useful information from the traces. Profiling a system's use combined with systematic organization of traces and snapshots of interfaces would benefit the users and ease their comprehension. Once a user gets to locate their desired functionalities for analysis they may also want to know about how that portion of the code behaves. Therefore, metrics will allow them to take a glimpse of the code's behaviour and analyse each feature individually. For instance CPU time and total execution of an invoked method can lead to findings of bugs, vulnerabilities or improper use of API in the code segment. A greater number of execution of an API for a particular code segment if compared over versions may provide an insight for focusing only on those ones with high priority. Similarly, too much CPU time consumption may indicate that the portion of code needs to be reprogrammed in an optimized way to reduce the time.

Let us consider two code segments a an b in Figure 4.6 having two different implementation. In Figure 4.6(a) the method gettingURLResponse() has been written in more verbose format showing every detail of the parameter declaration and object creation. However, in Figure 4.6(b) we can observe that the same method can be reimplemented in just one line of code. Both the functions give the same output but the only difference could be the time taken by them to execute. Therefore, CPU time metric value can indicate the users to switch the implementation of a method such as from Figure 4.6(a) to Figure 4.6(b).

The metric viewer thus will assist the users to understand the dynamics of the underlying code by providing more in-depth knowledge about a system. The viewer is populated with trace, method, group and active clone metrics when a user selects items from any of the main panel and the group panel. Since the main panel offers selection of items for the system use, tests, events and methods, the metrics at the metric viewer are displayed accordingly. Once the user selects any of the items such as an event from the main panel then the trace metrics: total number of package (N_p), total number of class (N_c), total number of invoked internal methods (N_{int}), total number of invoked internal methods (N_{ext}), total number of active clones (N_{ACLONE}), percentage of CPU time consumed by clone ($\%CPU_{METHOD}$), percentage of active clone file ($\%F_{ACLONE}$), and total number of active clone class (CC_{ACLONE}) and total active clone line of code (L_{ACLONE}) are populated in the viewer. Same metrics are also displayed when a user selects a system use or a test from the panel. However selecting any method within the event produces a different set of metrics

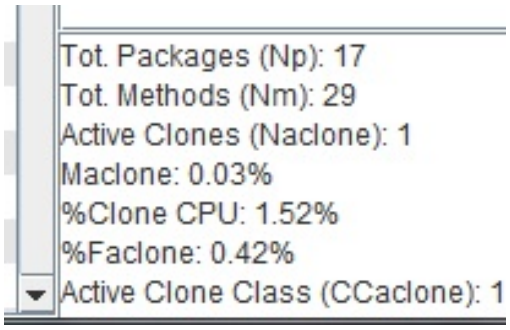

Figure 4.7: Diagram of the Metric Viewer

such as method breadth (Br_{METHOD}), total line of code for active method (L_{METHOD}), total CPU time consumed (CPU_{METHOD}), and total execution (E_{TOTAL}). These metrics will allow the users to identify the popularity of methods and APIs used in implementing a functionality. For instance a method being an active clone will let the users to understand how dispersed it is in the system and what is its impact.

The metric viewer is also populated with values when a user selects any item from the group panel such as 'Groups' and 'Active Clone'. The 'Groups' tabbed panel is enabled when a user selects external or both radio buttons to filter trace based on types. Similarly, the 'Active Clone' tabbed panel is activated when the user filters the trace based on internal type. Once it is enabled the users can navigate around the active clone classes. The panel supports both the single and multiple selection of classes. These selections update the traces in the main panel and at the same trigger display of metrics in the metrics panel. The metrics are clone class breadth, total number of active clone, total line of code and total CPU time.

Table 4.1: Summary of Features Supported by TrAM

Features	**Supported Features**
Search	Method
Segmentation	System use, test, event
Visualization	Real Image captured as screenshots, tree-like organization of traces
Filtering	External, internal, both
Code viewer	Method calls belonging to a particular segment (event)
System's Feature Implementation	Method calls and their call graphs
Metrics	System, trace, clone, method, groups
Type of subject system	Java
Categorization	External groups, active clones

4.5 Tool Comparison

Tables 4.1, 4.2 and 4.3 show the features supported by our tool, TrAM and its comparison with other trace analysis tools both in terms of supported features and their strengths for solving the requirements of the developers. To compare TrAM with other trace analysis tools we have considered the comparison into two facets: a) Feature based comparison and b) Scenario based comparison.

4.5.1 Feature Based Comparison

In the feature based as shown in Table 4.2 TrAM is compared with other trace analysis tools in terms of supported features and techniques. The criteria some of which are adapted from [28] and then extended for the tool comparison are as follows:

a. Trace Modelling

b. Trace Filtration

c. Trace Exploration

d. Trace Modularization

e. Categorization

f. Granularity

g. System Interface View

h. Trace Quantification

i. Trace to Source Code Mapping

Trace Modelling

Trace modelling is an important feature that is necessary to represent the traces in a meaningful and understandable manner by managing their excessive volume and size. Several ways are used to represent the traces such as a tree or a graph.

Trace Filtration

This feature is a complement of trace modelling and allows reduction or compression of the traces. The main intent of filtering is to remove some components of the traces in order to abstract out its content. In data collection, techniques are applied during the collection of traces both at the system or the component level. In the system level technique, a maintenance engineer is unaware of the methods used to develop a certain functionality. On the other hand, component level collection requires the engineer to know in advance about

the components they wish to instrument and analyse. Pattern matching technique involves the reduction of execution traces by grouping the similar patterns using a number of matching criteria such as setting a parameter or setting the depth at which traces should be compared. In sampling only a portion of the traces are selected for analysis using sampling parameters which pose threat when one setting may not work for another different sample. Architecture level filtering allows trace reduction by showing interaction among components at the architectural level instead of the objects and classes. However, the main drawback is that the technique is solely dependent on the presence of the system's architecture.

Trace Exploration

Once the traces are structured in the main panel using trace modelling techniques they must be made usable to the users using exploration techniques so that they can navigate through the traces. Trace exploration at the basic level offers the engineers to scan through the structure using clicks, search, browse, and animate the components of the desired interests.

Trace Modularization

Besides the immense volume of traces, another problem which the developers encounter while analysing them is interpreting the textual representations containing a series of method calls. Modularization of the traces can far ease the process of understanding them especially when one wants to know which system use the segments of traces belong to. Moreover, further modularization based on events can also add enhanced meaningful information to locate which user interface activity the traces belong to.

Categorization

Categorization allows the maintenance engineers to study the content of the traces by clustering the method calls generated during the execution based on groups and active clones. This feature lets the engineers to get a more focused view of the components of the traces.

Granularity

Granularity signifies the levels at which systems can be viewed for analysis. Object and class level granularities can assist engineers to view a system's implementation. Object level provides the scope for visualizing the interaction of method calls among the objects while class level on the other hand is a high-level representation of the classes implementing a particular feature.

User Interface View

Execution traces can be visualised using interaction diagrams such as graphs or trees. However, a live capture of the interface image on which an engineer is working can enhance the comprehension of a system's implementation further. Screenshots corresponding an event can let the users grasp the system design easily.

Table 4.2: A Comparison of Trace Analysis Tools Based on Features (taken from [28]) and then extended

Trace Analysis Feature		Shimba[92]	ISVis[37, 38]	Ovation[63]	Jinsight[62]	Program explorer[48]	AVID[94, 28]	Scene[44]	Collaboration Browser[70]	SEAT[30]	**TrAM**	Extravis[15]	Salah et al.[85]
Type	**Features**												
Trace Modelling	Tree Structure	✓		✓	✓		✓	✓	✓		✓		
	Graph		✓		✓	✓	✓					✓	✓
Trace Filtering	Data Collection	✓									✓		✓
	Architecture Level Filtering						✓						
	Sampling	✓					✓						
	Pattern Matching	✓	✓	✓					✓	✓			
Trace Exploration	Basic		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Trace Modularization	Segmentation by system use										✓		✓
	Segmentation by events										✓		
Categorization	Group										✓		
	Active Clone										✓		
Granularity	Object		✓	✓	✓	✓		✓					
	Class	✓	✓			✓		✓	✓		✓	✓	✓
System Interface View	Snapshots										✓		
Trace Behaviour Analysis	Metrics				✓					✓	✓	✓	
Trace to Source code Mapping	Source Code Viewer		✓							✓	✓	✓	

Trace Quantification

One of the ways towards understanding the behaviour of components of a system is through studying the traces which can be achieved using a set of metrics. Metrics calculated on the execution traces, can allow engineers to analyse various properties such as CPU time, execution, and so on of the components that implemented a specific feature.

Trace to Source Code Mapping

The source code view helps the users to visualize the original code of the system in conjunction to the image and the invoked methods. The mapping allows the users to compare the method calls in the traces with their actual implementation in the source code. This way they can analyse which methods have been called and which ones did not. The users can also get an overview of the various branching such as if-else or case statements within the method block.

4.5.2 Discussion

Table 4.2 shows the comparative features of the trace analysis tools in the form of matrix where the '✓' denotes the feature implemented by the tools. We can see that trace modelling using any of the tree structure or graph is adapted by all the tools with AVID an Jinsight using both of the techniques. Our tool TrAM adapats the tree structure because our intention was to let the users navigate through the entire execution trace. In case of trace filtering, pattern matching is a common feature implemented by Shimba, ISVis, Ovation, Collaboration Browser and SEAT. On the other hand, TrAM makes use of the data collection technique to collect traces at the system level and filter them based on segmentation both by system use and events. This provides more organized way of representing trace and enhances greater understanding towards feature location. Trace exploration is also supported by the tools regardless of the trace modelling techniques (tree structure or graph) and we can see that TrAM also incorporates the technique. Trace mining is further enhanced using the categorization feature by grouping external methods based on API and internal methods as active clones. TrAM has the capability to facilitate the users browse through the category and study group wise use of methods and their behaviour using a set of metrics. System interface view which provides live snapshots of the interfaces is only offered by TrAM which provides the advantage of reducing the cognitive load of the users for thinking what could the actual implementation would look like by viewing the trace. Trace to source code mapping is another added feature of the tools where the actual source code is also displayed so that the users get a glimpse of how a particular feature of a system is implemented which is supported by ISVis, SEAT, TrAM and Extravis.

4.5.3 Scenario Based Comparison

Comprehending a program's implementation is a challenging and a complicated process during software maintenance. It consumes an integral part of the maintenance activity. The better a system can be understood, the more precise and quick the changes can be made. Every tool has their own strengths and weaknesses and each of them even though is designed for understanding a system's implementation, has its own way of exposing the underlying details. However, the available tools need to be evaluated through comparisons among themselves. This will let the developers to select the right tools to use for the right purpose. The fact that the tools need to be evaluated is further aggravated due to the absence of any evaluation criteria or representative benchmarks. Since tools deploy different techniques, approaches and parameters, establishing a universal criteria is difficult. Therefore to overcome this difficulty and to compare the trace analysis tools more uniformly we have adapted a predictive, scenario-based approach which are as follows:

S1: A user wants to know the implementation details for a particular feature of a system and the events (thread signal, button clicks, display updates, mouse events and so on) associated with it. For instance, a user wants to know the method calls for a button click event such as the feature 'save file'.

S2: A user wants to know the implementation details for a particular feature of a system using interactions between objects and classes. For instance, a user wants to know the number of instances of objects created and exited for the invoked classes.

S3: A user wants to know the implementation details for a particular use of a system using execution patterns containing class and method calls. For instance, a user wants to know the call graph of the series of methods invoked for a particular use such as drawing a rectangle.

S4: A user wants to know more details about the runtime behaviour of a system using metrics. For instance, a user wants to know the CPU time consumed or number of times a method, or groups of methods invoked for a particular use such as drawing a circle.

S5: A user wants to know more details about the runtime behaviour of a system using metrics, view snapshots of the user interface and at the same time see the original source code of the feature.

S6: A user wants to know more fine grained details of the features by categorization into groups of application programming interface, internal methods or clone methods. For instance, a user wants to know details of an active clone class such that in what contexts the active fragments are used, times invoked, CPU time consumed and so on. Similarly, a user can also analyse the context of use and dynamic behaviour of groups of API such as 'String', 'io', 'Util' and so on.

Moreover, to qualify each scenario, the tools that are able to fulfil, we have set three types of qualifiers. Each qualifier describes the degree to which the tools are capable to meet the scenario requirement. The qualifiers are classified as 'can analyse', 'partially analyse' and 'cannot analyse'. The qualifier 'can analyse'

Table 4.3: Summary of Strength of Tools Based on Scenarios

• can analyse ⊙ partially analyse ○ cannot analyse

Tools	Scenario 1	Scenario 2	Scenario 3	Scenario 4	Scenario 5	Scenario 6
Shimba	•	•	⊙	○	○	○
ISVis	○	•	•	○	○	⊙
Ovation	⊙	•	•	⊙	○	○
Jinsight	•	•	•	⊙	⊙	○
Program Explorer	○	•	•	○	○	⊙
AVID	○	•	•	⊙	○	○
Scene	•	•	⊙	⊙	⊙	⊙
Collaboration Browser	•	•	⊙	○	○	○
SEAT	○	○	•	⊙	•	○
TrAM	•	⊙	•	•	•	•
Extravis	•	•	•	⊙	⊙	⊙
Salah et al.	⊙	•	⊙	⊙	○	○

signifies that the tool satisfies all the conditions of a scenario and provides a complete details of the system according to the user requirement. 'Partially analyse' means that a tool satisfies a few of the scenario conditions but in a different way without fulfilling all of the conditions. This means that few things such as parameter, values, or other information relevant to the scenarios are missing. However,'cannot analyse' means that the tools infrastructure does not allow the users to look for what they want in order to understand a system.

Table 4.3 depicts the specialities of the tools on the scenario basis as described by the qualifiers. For scenario 1 all the tools support the analysis with the exception of Program Explorer and SEAT. Ovation and the approach proposed by Salah et al. can partially allow the users to analyse a system's feature implementation by classifying the traces into segments by event types. We can observe that most of the tools available so far are designed for the scenario 2 as well. Scenario 2 is a situation where users want to know a system's details using interactions at the class or method levels. An interaction means communication between two components such as class or method. All the tools except TrAM and SEAT qualifies for this purpose. The interactions among the components are represented using diagrams with horizontal lines, vertical lines and other notations using boxes and arrows. However, TrAM produces a call graph of method calls from which the interaction among the classes such that the caller-callee dependencies can be derived. Knowing the method calls and the pattern of their execution also provide added information towards program

comprehension. How and when the methods are called and what patterns do they follow lets the users get more insight about the flow of program execution. Interestingly, from the literature survey we also came across that all the tools have the ability to solve the users' purpose for the scenario 3 except Shimba, Scene, Collaboration Browser and Salah et al. where they do provide inter-class information but fail to assist in showing the execution patterns.

Metrics are one of the important means in making the comprehension process more informative by quantifying the dynamic behavioural aspects of a system. This is represented as scenario 4 and the majority of the tools implements the basic dynamic metrics such as total number of calls and CPU time unlike TrAM. TrAM implements six kinds of metrics categories such as system metrics, clone metrics, group metrics, active clone metrics, method metrics and trace metrics. Scenario 5 further complements the comprehension process by letting the users to locate the features more precisely and explicitly. Besides providing information through metrics a live view at the snapshots of the user interfaces and the corresponding original source code at the same time will assist in pin pointing the nature and the portion of the code of a feature in the system accurately. SEAT and TrAM are fully capable of fulfilling this with Jinght, Scene and Extravis are partially able to handle the situation such that none of them displays the snapshots of the user interface. Finally, scenario 6 portrays a situation when a user might want to further focus on their analysis on particular methods, API or similar fragments such as clones. Only TrAM qualifies for this scenario and it provides an extended feature of letting the users to manipulate the API, internal methods and clones to gain insights about their implementation details and associated dynamic behaviour. Few tools such as ISVis, Program Explorer, Scene and Extravis do provide focused view through categorization partially.

In summary, we can conclude that the majority of the trace analysis tools are designed for analysis of a system with emphasis on the interactions of the modules such as class and methods and the execution patterns of the method calls. Some of them also highlight the use case scenarios and give insights to the context of use of a system. If we walk from scenario 1 to scenario 6 the tools gradually fail to reveal the dynamic behaviour of a system in terms of metrics and also fall behind in specifying the snapshots of the user interface of the system being analysed with the underlying source code. They also lack the features of enhanced information browsing for methods, API and clones through filtering and categorization. On the other hand, TrAM implements the necessary features that fulfils all the scenarios with an exception of scenario 2 where it does not offer complete analysis of the interactions between the modules.

4.6 Comparison with Profilers

In this section, we also provide a comparison of TrAM with other profiling tools. Since each tool has its own strengths and weaknesses, a comparative study we provide a deep and clear insight about its specialized area. We have considered both UI based as well as command line based profilers. JRat, JProfiler, JIP, and JMP are the UI based profilers while DJProf, HPROF, and gprof are the command line based profilers. For

Table 4.4: A Summary of Comparison of TrAM with Profilers

Profilers	Feature Identification	Metrics				Call Graph	UI Snapshot and Code View	Categorization	
		Trace	Method	Active Clone	Group			Method Graphs	Clones
JRat	No	No	Yes	No	No	Yes	No	No	No
JProfiler	No	Yes	Yes	No	Yes	Yes	Yes	Yes	No
JIP	No	Yes	Yes	No	No	Yes	No	No	No
JMP	No	No	Yes	No	No	Yes	No	Yes	No
DJProf	No	No	Yes	No	No	No	No	No	No
HPROF	No	No	Yes	No	No	No	No	No	No
gprof	No	No	Yes	No	No	Yes	No	No	No
TrAM	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

comparison we have considered five categories: feature identification, metrics, call graph, UI snapshots & code view, and categorization. From Table 4.4, we can notice that all the tools excepting TrAM fail to locate features which mean they do not provide insight into which portions of code are responsible for a particular feature. Although method calls are represented, they do not provide the context in which they are used. TrAM satisfies this requirement through segmentation of trace both in terms of system use, tests and events. The system use signifies the purpose of use of a system such as drawing objects, tests signify types of use such as drawing a rectangle, and events signify user interactions with the interface such as button click. In case of metrics, we have considered our taxonomy of metrics trace, method, active clone and groups which we proposed.

If we observe closely, we can find that all the tools are capable of calculating the method metrics such as total execution, and total CPU time. For trace metrics, TrAM calculates all the metrics and provides detail information about trace such as total package, or class executed, total files invoked, and CPU time based on system use, test and event. However, JProfiler and JIP still fail to provide the CPU time information based on system use, test and events. Active clone metrics is a new induction into our tool TrAM where we mine dynamic information of similar code fragments from the trace. Therefore, only TrAM incorporates the feature of active clone mining and provides more enhanced facility for trace mining. Similarly, JProfiler and TrAM are the only tools that can calculate the group metrics such as CPU time per group, number of active methods in a group, number of tests in which a group is active. However, JProfiler fail to distinguish the existence of group in terms of system use, test and event. Call graph representation is an important feature that shows how and in what sequences the methods in a system have been called. All the tools except DJProf and HPROF provide the call graph. One of the main contributions of this thesis is understanding a systems feature using method calls and each method has a block of code associated with it. The UI Snapshots & code view feature helps the process of understanding about how a feature is implemented by linking the method

calls with the source code. Only JProfiler and TrAM supports this and with an exception of JProfiler where it does not incorporate the UI snapshot. Finally, in case of categorization, JProfiler, JMP and TrAM can categorize methods based on groups while in case of categorization through clones only TrAM has the ability to fulfil it.

4.7 Tool Evaluation

To evaluate the tool we conducted experiments in two phases. The first phase included general evaluation by a number of users for their opinions and is subdivided into two sections: a) general feedback and b) task based feedback. The second phase included the actual validation of the outcomes of the tool mining data by the expert developers.

General Feedback

In this part of the user study, we have set some general questions in order to get feedbacks on the overall opinions of the users based on their familiarity with software development and various program comprehension tools they have used. One of the main motivations of this thesis is to build a tool support for understanding the implementation details of a system. In this part of the user study, our intent was to get the feedback from the participants about their opinions towards different ways of understanding a system and the tools they have used to support their comprehension.

From the feedback where 93% of the participants had the experience on modifying or fixing or optimizing a system developed by other developers, we have come across that when they were asked *whether they wrote any documentation for a system they developed*, 14% answered "no", 29% said "partially" while 36% and 21% replied "yes but did not maintain the documentation" and "yes and maintained the documentation" respectively. On the other hand, 64%of the participants also did not attempt to use any documentation tools such as doxygen or javadoc. To further verify their feedback they were asked, *if they were given a system and asked to modify some of its functionalities, then what would they do initially to understand the code?* Interestingly, we came across the fact that 31% were in favour of reading documentations, 22% would ask the developers to help them out, 22% would run the system and 20% would use an automated tool to them out. However only 4% would try to create a new documentation of the existing system. To support this we found that 43% of the participants have experience with profilers or debuggers out of which 38% of them used to find errors in code, and 31% to understand a specific portion of code. From here we can conclude that majority of the users were in favour of using documentation followed by other techniques such as discussing with the developers, run the system or use other automated tools.

Task Based Feedback

In this section we will describe how the tool is evaluated based on its usefulness. Generally, the usefulness of a tool is determined by the following criteria:

C1 *How effective our tool is in order to meet users' requirements during the program comprehension?*

C2 *How simple and easy the tool is to operate for source code analysis?*

C3 *How presentable the information is in the tool for the users to interpret and analyse?*

We then further conducted an intensive study on three different systems developed in our lab. The results were presented to the actual developers and their opinions were noted.

4.7.1 User Study

Our main goal is to evaluate the overall usefulness of the tool and for that we had to design a methodology to conduct the study. The main challenge in this study was availability of the users and the type of users we had to consider. To fulfil the experiment we have conducted a series of steps starting from experimental setup to analysis of the result.

Experimental Setup

To evaluate the usefulness of our tool we required a set of participants, a questionnaire, methods of evaluation, and analysis of the result. We also selected an interactive user interface based subject system to present the participants with the implementation details to comprehend a system. The following are the steps that constitutes our setup for the experiment.

1 **Design a questionnaire**: We have designed a total of 24 questions to cover responses of the participants from different perspectives. The questions in the questionnaire are classified into different sections. The first section of the questionnaire is designed to acquire the background of the participants with respect to their knowledge about the domain of software development. The second section extracts information about the common scenarios that they have faced and how they have dealt with them. This part also allows them to do specific tasks using the tool and answer the related questions. The questions in this section are solely designed to get user satisfaction level. The satisfaction level is represented using a likert scale. Five different levels such as strongly *agree*, *strongly agree*, *neutral*, *disagree* and *strongly disagree* defines how satisfied the participants are about the tool's usefulness after the tasks are done. Therefore, as a part of the questionnaire design phase we have also put forward a set of tasks for the participants.

2 **Demo session**: It is the second phase of our experiment modelling which initiates the user participation for the study. The users were given introduction about the nature and purpose of the study. Then they

were briefed about the tool that they would use and the motivations behind it. We then ran the tool over a subject system and presented every feature of it. We also described the architecture of the tool including the visualization and the information management in particular so that the participants can operate it easily. The participants were then handed with the tool and allowed to operate for a while to get familiar with it.

3 **Task Performance**: After the demo session the questionnaire was handed to the participants. Each individual task was followed by a question that recorded the participants' opinion based on their level of agreement on the performed tasks. It was an open discussion session and the participants had the liberty to ask questions regarding any difficulty they faced during the task completion. All the participants had to answer the questions on-line. We selected 14 participants with backgrounds in software development. The study session was carried on Windows 7 workstation on which the TrAM was installed as a client environment. The client side was connected with the database server to fetch data from it and complete the mining process.

4.7.2 Task Design

To evaluate our tool and address the three evaluation criteria we have set four tasks that will allow the participants to express their opinions accordingly. Each task intends to gather statistics about the usefulness and the usability of the tool. Usefulness signifies how the tool can bridge the gap between the complexity of a system's implementation and the program comprehension by the users. Usability portrays how simple and easy the tool is to operate and therefore assists the users to interpret the result during the comprehension process. Table 4.5 provide a summary of the task the users were provided with.

Table 4.5: Summary of Tasks

Task	**Description**
Locate feature	The participants were provided with a scenario and then were asked to find the portion of code that was responsible for that particular task.
View Statistics	The participants were asked about a scenario such as how the behaviour of a particular method can be found from the trace.
Filter methods	The participants were asked about a scenario such as how specific information about methods can be obtained using the tool.
Categorize groups	The participants were asked about a scenario such as how a group wise information can be obtained.

To address the evaluation criteria we have set different questions specific to those criteria. The participants were asked questions on usefulness, usability and the design of the tool. The following are the sample questions that are designed for each criteria.

C1: How effective our tool is in order to meet users' requirement during program comprehension?

1 *It was very difficult to isolate the feature given a bug using the source code.*

2 *The tool clearly presented the statistical information of the selected methods.*

3 *The tool very easily allowed to retrieve the information of active clones in the system.*

4 *The filtering mechanism was very quick to filter out the methods that was required.*

C2: How simple and easy the tool is to operate for source code analysis?

1 *What difficulty did you face in using the tool?*

2 *What features do you think are the most significant in the tool that guided you towards your goal?*

3 *How satisfied are you with the filtering by type, searching, categorization by groups, categorization by active clone class, image and code viewer and metrics view?*

C3: How presentable the information is in the tool for the users to interpret and analyse?

1 *What portion of the tool you think requires improvement?*

2 *The information in the tool was well organized.*

3 *Overall the tool was very satisfying.*

4 *The overall performance (delay when a user clicks on any item) of the tool was very good.*

4.8 Findings

After collecting all the data from the 14 participants, we analysed them to refer the three evaluation criteria. Our first criterion of evaluation was based on effectiveness of the tool in terms of fulfilling 4 tasks performed by the participants. For the Task 1 we asked the participants to locate a feature given a bug both with or without the tool. Interestingly, 92% of the participants expressed positive agreement that the tool made it very simple and easy to identify the feature while 75% had difficulty in locating the feature using the source code only. Task 2 was a bit different where they were asked to find the statistics of a particular method. All the participants were able to complete it and strongly agreed what the tool offered them with. This indicates that the hierarchical tree-like structure made the task so simple that only a click allowed them to see the

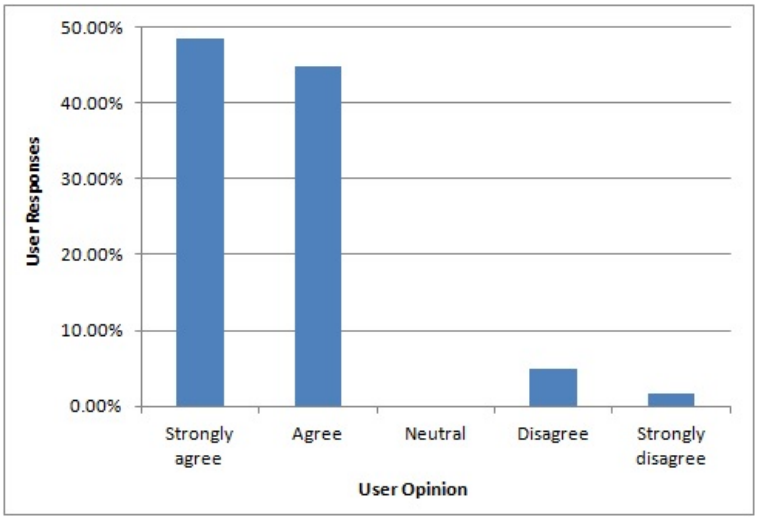

Figure 4.8: Effectiveness of TrAM to Meet Users' Requirement During Program Comprehension

statistics of the method in the metric viewer. Task 3 was also dealt with feature identification but from different perspective. This time they were asked to find the similar fragments and observe their properties. 64% of the participants were strongly familiar with the term code clone and 36% had very little idea about it. 58% of the participants were able to operate and interpret the output of the active clone classes. The last task was made to test the performance of the tool in terms of filtering. Since we adapted the hierarchical tree-like structure each root node represents a test, each test encloses multiple events and each event groups multiple methods. The structure may become immense in size with abundance of texts. Each event inside a test node is associated with snapshots. This might put constraint on memory utilization and cause poor system performance. Switching between the external and internal filtering may allow the user to judge how quick the tool responds to the request. 75% of the participants were very satisfied with the performance and strongly agreed that the tool was very quick to filter out the methods. Figure 4.8 shows the overall summary of the users' response. We can see that almost more than 45% of the participants agreed that the tool was very effective in accomplishing the task they were given to. However, we also got a few exceptions where less than 10% of the participants disagreed with the fact.

The second part of the evaluation criteria was to determine how simple and easy the tool was in terms of operation. Here the participants were asked for their opinions about the controls of the tool and whether they assisted them to solve their purpose while performing the tasks. Figure 4.10(a) shows that 33% of the participants voted for the feature *image* and 31% was in favour of the *navigation through tree-like structure* followed by filtering by group and filtering by method type. On further investigation to the response of 11% of participants we noticed that it was there industrial experience in using other profiling tools that made their task easier. However when asked for difficulties they faced in using the tool 36% of the participants in Figure 4.10(b) pointed out about the complex organization of the architecture. This may be due to the fact that the main panel provides a complete view of the trace in a two dimensional format in a hierarchical

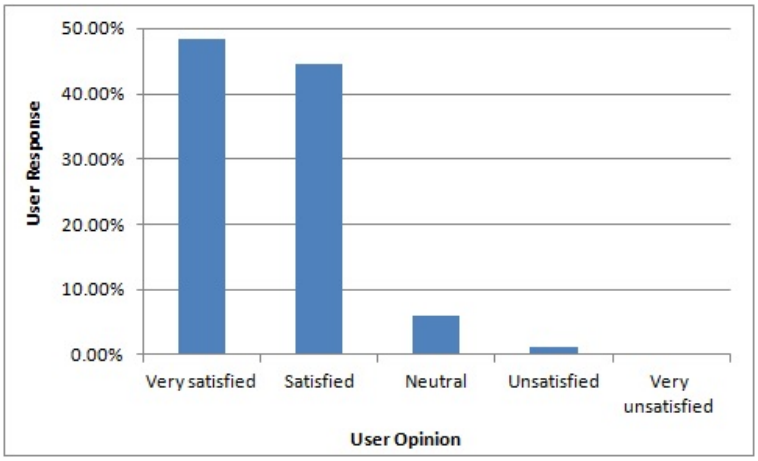

Figure 4.9: User Response for the Difficulty of Feature Use

structure. The y-axis shows the values of the components. At some point a situation may arise when too many nodes are open making the panel overcrowded with textual information. If we observe Figure 4.9 which is an overall response of the participants, we can find that majority of the participants were comfortable using features of the tool for performing the task with few exceptions of participants below 10% who were neutral and unsatisfied with them.

We wanted to know from the participants about the design implications of TrAM for the controls and the visualizations. Figure 4.11 shows the overall responses of the participants on individual features of TrAM. More than 50% of the participants were very satisfied with categorization by groups, active clone class denoted by catg[group] and catg[ACC] in Figure 4.11, and the image and code viewer feature. It is obvious that the participants were interested in knowing the locations and the context of the external methods in the code. Moreover, categorization with active clone class also looked important to them because knowing the similar code fragments can allow them to visualise how active clones are dispersed in the system. Image and code viewer had the highest satisfaction level with over 70% because they were able to map the underlying code with the live snapshot of the interface. This prevented them from thinking about, what and how the interface could have been for that specific feature, but instead let them explore the implementation specifics of the features in one place.

To gain more insight about the design another feedback was collected on the level of guidance of the features to users towards their goal.

Finally, the participants were asked about how presentable the information in TrAM was to interpret and analyse the result which is shown in Figure 4.12. If we look at Figure 4.12 we can see that more than 40%of the users strongly agreed that the information was well organized. They also agreed that the tool provided better performance in terms of delays during the operation of tasks. Filtering, searching and thumbnail loading were considerably fast as well.

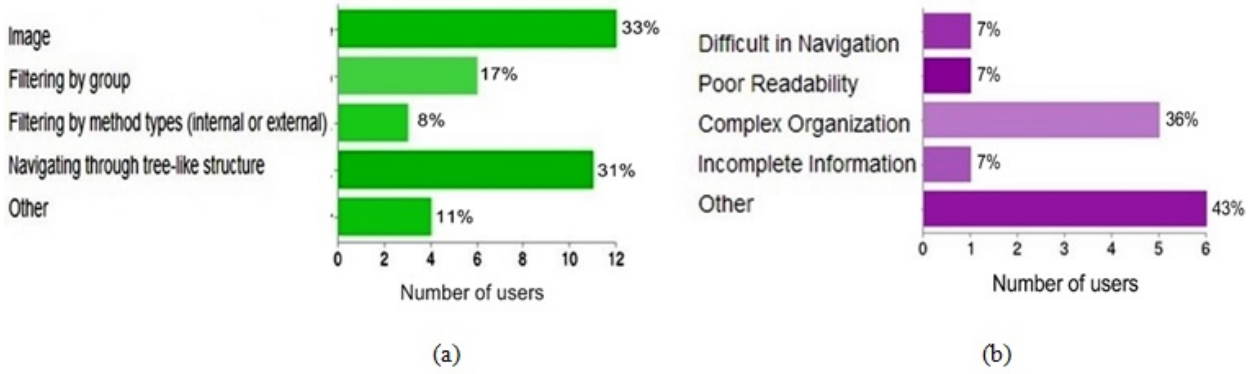

Figure 4.10: User Response for the TrAM Features (a) Most Popular Features of TrAM and (b) Features Requiring Improvement

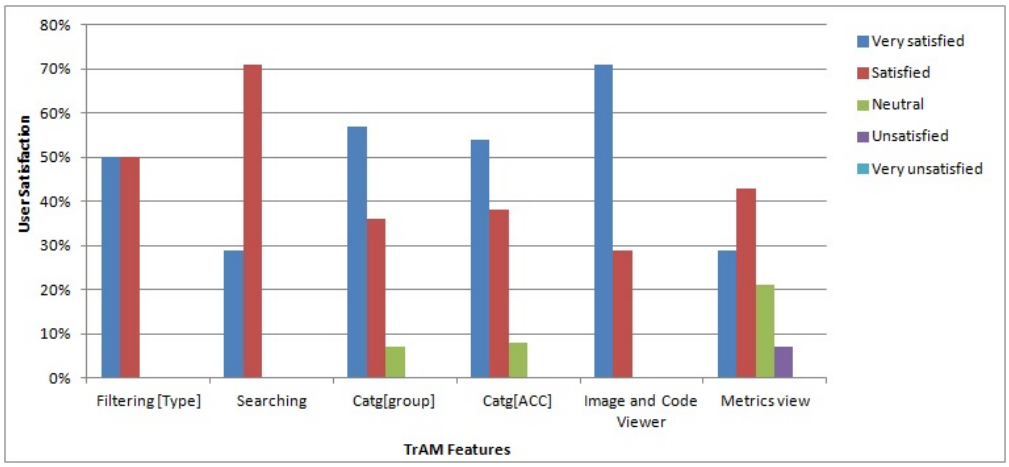

Figure 4.11: Comparison of User Responses for the TrAM Features

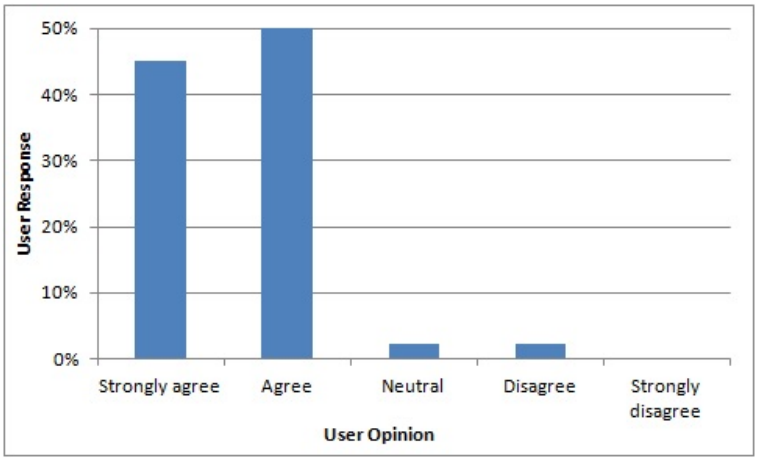

Figure 4.12: Presentation of Information in TrAM for the Users to Interpret and Analyse

Table 4.6: Summary of the Features Analysed

System Name	Line of Code	Total Files	Features
TrAM	7799	66	Search
			View image detail
			View event statistics
VisCad	17772	129	View metrics
			View treemap
			Browse code
Visualization Tool	19344	119	Search rules
			View method by coupling
			View method by co-change

4.9 Expert Opinion

Our studies in the previous section was to determine the overall usefulness of our tool in software maintenance and how usable the tool is in terms of features. This marked the first phase of our evaluation. However, the question remains on how accurate the tool is in terms of feature identification and statistics calculation. Locating the correct feature, mapping the exact portions of the source code, calculating the metrics values and categorizing them by active clone class and external methods contributes to what extent and effectively the tool can be relied for the maintenance activity. This demands the opinions of the users who are expert in their domain and have developed at least a system using java. One of the main drawbacks in experimenting with open source systems is that we are not aware of the domain and not even familiar with their implementation details. Therefore, only expert opinions of the developers about their own system can assist us to evaluate the output of our tool.

To begin with we have chosen three subject systems developed in java by developers in our lab. Table 4.6 provides summary of the systems. Before proceeding with the trace collection phase, we first discussed with the developers about the key features of their system. We assume that there are always certain key features in systems that are most frequently used by the users and therefore, requires attention during maintenance activity. We then collected traces for those key features and mined them for the implementation details. The results were presented to the users and asked to verify them. As part of the evaluation phase three key features of each system were identified and selected for the trace collection. For each system we presented them with active clone classes, tests, events, images, underlying codes and metrics. One of the main aspects of our trace mining approach is method, clone and API mapping. If the tool is able to map them correctly then it can be inferred that portions of the source code will be located correctly as well. Similarly all other mining that are dependent on the mappings would be achieved accordingly. The tool was provided to the developers to verify the outputs. For all the three systems, each developer was satisfied with the results specially the image to code mapping for a particular feature. The tool was able to locate the exact portions of the source code responsible for each of the feature. Next, they were asked to verify the categorization of trace based on external group and the active clone class. Since they were aware of the clones in the system

then they also knew which portions of the code they have copy pasted during the implementation. After inspection of the active clones they were strongly satisfied with the output too. They were also able to observe the source code to identify the location of the active clones and to what context the cloning has been done.

Chapter 5

Source Code Clones at Runtime: An Exploratory Study

5.1 Introduction

Code duplication is a common programming practice. Programmers use code duplication to increase the speed of the software development process by using similar code to implement common functionality. Code duplication can also reduce the risk of errors, particularly if the copied code has been previously tested and proven. Similar code fragments are called 'clones' and the process is known as 'cloning'. Different clone granularities are studied, such as method clones (clones that are methods or functions) and block clones (clones a few lines in length). In this paper we focus on method clones. Fragments that are not exact copies of each other, but share a certain level of similarity are known as near-miss clones, in which statements might be added, deleted, and/or modified in the related fragments. Previous studies revealed that duplicate code is present in software systems in amounts ranging from 5-15% [74, 73, 72, 104] to as high as 50% [71]. Although cloning is an inexpensive and often productive practice, it introduces complications during software maintenance. A code fragment may contain a bug, and when copied elsewhere, the bug is also propagated. As well, a bug may emerge when a clone is used in a new context. Fixing a bug in one clone may require reviewing and perhaps applying a similar fix to its clones. Any inconsistency in making a change to a set of clones may introduce new bugs. For large software systems and depending on the extent of the cloning, such maintenance effort may be expensive. Despite the ongoing debate as to whether code clones are harmful or not [4, 23, 39, 40, 52, 60, 59, 58, 57, 56], it is believed that clones are unavoidable, and not all clones can or should be removed [43, 82, 102].

Clone management is an active research area that has contributed a diverse set of tools and algorithms to aid in the management and maintenance of software. There are also a few novel studies that introduces the notion of higher level clone management [69] or finding out an optimized scheduling of the candidate clones for refactoring [101, 103]. However, clone research has not well addressed the runtime implications of source code cloning. Moreover, clone researches are rich in detecting similar fragments syntactically but the study to reveal the context in which they are used is still missing. In this paper we investigate source code clones at runtime, referring to clones as 'active clones' if they are invoked when a software system is in use. For

example, if a particular use u of a system results in a clone c being invoked, we say that clone c is active with respect to use u. From this definition and given a set of uses $\{u_1, u_2, ...\}$ and clones $\{c_1, c_2, ...\}$ we are able to identify the extent clones are active at runtime, analyze active clone resource use (e.g., CPU time) and define and calculate a set of active clone metrics. Associated with a particular use u of a system, is an execution trace, which we denote as t_u. An execution trace t_u is a sequence of method calls $\{(1, m_1), (2, m_2), ...\}$ and so $codomain(t_u)$ is the set of method calls invoked during the use (i.e., $codomain(t_u) = \{m_1, m_2, ...\}$). Our focus is on method clones, and so if method clone $c \in codomain(t_u)$, then c is an active clone with respect to use u.

Similarly a clone class that includes at least one active clone fragment is referred to as an active clone class. For example let us consider three clone classes, $c1$ with clone fragments (a, b, c), $c2$ with clone fragments (d, e), and $c3$ with clone fragments (f, g, h). Let us assume that using our approach, we found that only g is the active clone fragment (i.e., the corresponding method was invoked in the trace). Because $g \epsilon c3$, $c3$ with fragments (f, g, h) will be the only active clone class. Execution traces may be collected in the field, however for our study we used a set of tests to collect execution traces. We consider a test to represent a typical use of a software system by an end-user, or a typical use of a software system by a developer in support of a software development or maintenance task. Software developers often use testing to execute a software system with the intent of locating errors, detecting faults and verifying or validating a system's qualities and functionality. If a developer runs the system or designs a test suite for the task at hand, they can use our approach to identify active clones related to the tests and consequently to the task at hand.

It is clear that the choice of tests effects the extent active clones are discovered. For example, a test suite may be specifically designed for full coverage and so in such a situation one would expect all clones to be active. Our interest, however, is in tests that correspond to a particular use of the system. For example, tests that are specifically designed to exercise a single feature, requirement or use case scenario. Although a test suite may exercise the entire system, a single test typically is designed to exercise a subset of the system. A clone, then, is active with respect to a use, a set of uses, a set of tests, or a test suite. It is determining the method clones that are active when a single test or a small set of tests are run. that we focus on in this paper. As a result, we are interested in addressing the following three research questions:

RQ1: *To what extent are clones active during runtime?*

RQ2: *How active are the active clones?*

RQ3: *Does active clone identification support software maintenance activities?*

5.2 Finding Active Clones

After running the system with a specific set of test cases and then collecting its dynamic behavior in the form of trace we perform the clone mapping phase. This step maps the trace information to the codebase.

Table 5.1: Subject Software Systems

Domain	System	No. of Java Files	Total Size	Version
Text Editor	RText	319	5.76M	2.0.0
Graphics	JHotDraw	309	20M	7.0.6
Database Server	HSQLDB	513	14M	2.2.6
Point of Sale	UnicentaoPOS	486	30.4M	2.5.0
Text Editor	jEdit	571	25.5M	4.4.2

Table 5.2: Sample Tests

System	Test	User Interaction
HSQLDB	Select	select * from testtable;
	Insert	insert into testtable values ('Current', 22, '2003-11-10', 18, 3, 'my name goes here'), ('Popular', 23, '2003-11-10', 18, 3, 'my name goes here'), ('New', 5, '2003-11-10', 18, 3, 'my name goes here'), ('Old', 5, '2003-11-10', 18, 3, 'my name goes here');
	Update	update testtable set (astring, firstnum, adate, secondnum, third-num, aname) = ('Older', 5, '2003-11-10', 18, 3, 'my name goes 'here) where astring = 'Old'
	Delete	delete from testtable;
	Create table	create cached table testtable (aString varchar(256) not null, first-Num integer not null, aDate date not null, secondNum integer not null, thirdNum integer not null, aName varchar(32) not null);
JHotDraw	Circle	User selects Circle and draws the object.
	Rectangle	User selects Rectangle and draws the object.
	Triangle	User selects Triangle and draws the object.
RText	Edit Text	Write in editor, Save a file, Edit text, Replace text and Find text in files
jEdit	Edit Text	Write in editor, Save a file, Edit text, Replace text, Find text in files, Print File, Close Application
uniCentaoPOS	insert	Adding customer data
	update	updating customer data
	delete	delete customer data
	report	generate customer report

For each method in the trace, if it contains a clone in the codebase, the corresponding clone and its clone class are marked as 'active'. This step determines which clone classes are active for that particular test case. First, using dynamic analysis we record execution traces for each individual test case. Each trace record in a trace file consists of the full class name, the class type, the method name, and the parameter types. Second, we obtain all the clones of the subject system detected by the NiCad clone detector. Third, we determine whether the trace related code contains any clones. For clone mapping, we created a script to extract all the method names from the NiCad XML file. The clones for which we get a match in the trace are called active clones, and then we retrieve the corresponding clone classes to which the active clones belong. These clone classes are treated as active clone classes.

Table 5.3: Method Clone Metrics

Systems	Files			Lines of Code			CC	N_{CLONE}	M_{TOTAL}	$\%M_{CLONE}$
	F_{TOTAL}	F_{CLONE}	$\%F_{CLONE}$	L_{TOTAL}	L_{CLONE}	$\%L_{CLONE}$				
RText	319	81	25.3	95036	3019	3	80	189	3698	5.5
JHotDraw	309	183	59.22	56420	7565	13.4	231	656	3260	20.1
HSQLDB	513	293	57.11	227545	20467	8.9	503	1323	9825	13.5
jEdit	571	161	28.2	115246	7320	6.4	212	419	7428	5.6
uniCentaoPOS	486	174	35.8	43032	4578	10.6	163	285	4260	6.9

Table 5.4: Test Based Metrics for Active Clones

System	Use Set	Test	N_{int}	N_p	Method Clones				
					N_{ACLONE}	L_{ACLONE}	F_{ACLONE}	$\%M_{ACLONE}$	$\%CC_{ACLONE}$
HSQLDB	Search	Select	461	89	40	545	22	8.6	6.8
	Data Entry	Insert	441	87	35	492	22	7.9	6.2
	Admin	Update	441	89	35	492	22	7.9	6.6
		Delete	441	87	35	506	25	7.9	6.6
		Create	442	90	35	492	22	7.9	8.0
JHotDraw	Draw	Circle	798	189	58	654	27	7.2	19.9
		Rectangle	797	189	56	639	26	7	19.5
		Triangle	819	195	60	645	28	7.3	19.5
RText	Compose	Edit text	591	153	11	265	7	1.8	11.25
jEdit	Compose	Edit Text	3220	310	220	400	44	5.2	37.7
uniCentaoPOS	Search	Select	101	37	5	43	5	4.9	3.2
	Data Entry	Insert	373	85	17	141	15	14.9	9.2
	Admin	Update	76	32	4	55	4	5.2	0.8
		Delete	68	28	4	30	4	5.9	0.6

5.3 Study Approach

In this section, we provide a brief description of the subject systems being studied, our approach and the tests used. All the tests were run on an Intel(R) Core (TM) i7 CPU @2.93 GHz with 4GB RAM. Table 5.1 provides a summary of the systems that were part of our experiment. To bring variation to our study we considered systems from three different domains: text editing, graphics and database management. We also selected systems that differed in size, ranging from 5.76M to 30M bytes of code.

In the first step of the static analysis phase, we use NiCad as the clone detector to detect near-miss clones both at the method level using consistent renaming at a 30% threshold. We use NiCad's XML output, which reports clone detection results with similar code fragments grouped into clone classes.

The reports from the clone detector were further analyzed to calculate the system and clone metrics using a clone analysis and visualization tool called VisCad [3]. NiCad's clone detection results and the codebase of the subject systems were used as inputs to VisCad. VisCad assists in calculating a number of metrics in response to user requests and produces interactive visualizations through which a user can investigate code clones at different levels of abstraction.

Table 5.5: Lines of Code and Number of Files Associated With Traces, Clones and Active Clones

Metrics	System				
	HSQLDB	**JHotDraw**	**RText**	**jEdit**	**uniCentaoPOS**
$\%L_{CLONE}$	3	13.4	8.9	6.4	10.6
$\%L_{ACLONE}$	0.24	1.2	0.31	0.71	0.31
$\%F_{CLONE}$	57.11	59.22	25.3	28.2	35.8
$\%F_{TRACE}$	17.3	61.2	47.9	51.9	17.5
$\%F_{ACLONE}$	4.2	9.1	2.3	7.4	3.1

To collect the traces for each system we designed tests for each user interaction. A subset of the tests we used are listed in Table 5.2. Traces for each test is recorded in text files for subsequent post-processing and quantitative analysis. For each test the trace generation is triggered by a set of defined inputs also shown in Table 5.2.

In this section, we present the results of our experiment identifying active clones in five different subject systems of varying sizes. The study and analysis of each system is presented separately. The research questions are discussed based on the experimental results shown in Tables 5.3 to 5.8. Table 5.3 provides metrics for clones, clone classes, files associated with clones, and lines of code for methods. Table 5.4 provides statistics for each system in terms of active clones and their classes while Table 5.6 lists the percentage of active clones for each type of user interface event by test and also provides CPU usage for each type of user interface event along with the percentage of time associated with the active clones. Table 5.5 shows the percentage of active clones for each test, indicating that clones are involved in each test. It also provides lines of code metrics for each system, comparing the percentage of clone lines with the percentage of active clone lines. This helps illustrate that fewer clones need to be considered when only considering active clones. Table 5.5 also shows file metrics for each system, comparing the percentage of files containing clones with the percentage of files invoked in the traces and the percentage of files containing active clones. Again, this shows that fewer files need to be considered when focusing on clones that are active during runtime.

One challenge during software maintenance involving clones is to determine which clones to look at first. Our study provides information that can help prioritize clone inspection. The tests are intended to correspond to a maintenance task, and so there is a high chance that clones contained in the implementation of the user interaction being tested will be executed at some point during the test. If a typical test does not yield active clones, one may not need to worry about clones for that particular maintenance task. However, in all cases and for relatively simple tests we were always able to identify active clones. For example, in the case of HSQLDB, 13.5% (Table 5.3) of the system's methods contain clones and we found that on average 8.04% (Table 5.4) of the invoked methods ($\%M_{ACLONE}$) contained active method clones. The average lines of code and files that are active is 0.24% and 4.2% (Table 5.5) while at least 6.5% (Table 5.4) of clone classes (CC_{ACLONE}) are active for each of the tests (i.e., out of 503 clone classes 33 clone classes are active).

Table 5.6: Active Clones by Type of User Interface Event for Each Test

System	Test	UI Event Type	UI Traces	MI	%AC	CPU Time (ns)	
						Total	%AC
HSQLDB	Select	mouseClick (JButton)	3	191	6.8%	8.31E+10	14.8
		threadSignal	50	1555	5.2%	2.86E+12	47.6
	Insert	mouseClick (JButton)	2	203	6.4%	7.31E+11	14.8
		threadSignal	36	1076	7.2%	1.86E+12	51.26
	Update	mouseClick (JButton)	2	192	6.8%	3.16E+11	3.12
		threadSignal	28	1160	7.2%	2.93E+12	44.44
	Delete	mouseClick (JButton)	2	204	6.4%	3.76E+11	3.79
		threadSignal	28	885	7.9%	1.93E+10	42.5
	Create	mouseClick (JButton)	3	196	11.6%	2.05E+11	4.86
		threadSignal	51	1164	7%	1.02E+12	3.12
JHotDraw	Circle	mouseClick (mouse event)	17	397	6.8%	1.66E+12	25.8
		mouseClick (Button)	12	89	3.3%	1.34E+10	0.47
		threadSignal	1	490	6.1%	9.60E+11	33.8
	Triangle	mouseClick (mouse event)	13	375	6.9%	6.03E+11	17.02
		mouseClick (Button)	12	89	3.3%	8.60E+10	3.79
		threadSignal	1	490	6.1%	1.29E+12	57.09
	Rectangle	mouseClick (mouse event)	13	399	6.0%	1.73E+11	7.7
		mouseClick (Button)	12	89	3.3%	6.03E+10	3.92
		threadSignal	1	490	6.3%	1.00E+12	67.18
RText	Edit text	mouseClick (JButton)	1	18	2.2%	8.00E-3	0.001
		mouseClick (JMenuItem)	5	18	1.6%	2.50E+6	0.005
		displayUpdate	30	124	3.2%	3.65E+7	0.080
		threadSignal	11	79	2.5%	4.49E+10	15.5
jEdit	Edit text	mouseClick (Timer)	5	18	1.6%	2.39E+12	9.9
		displayUpdate	30	124	0%	4.83E+11	0
		threadSignal	11	79	2.5%	2.22E+13	85.2
uniCentaoPOS	Select	mouseClick (JButton)	2	71	7.0%	7.43E+10	44.8
		mouseClick (Timer)	50	1555	0%	2.86E+12	0
	Insert	mouseClick (JButton)	4	203	6.4%	1.61E+11	43.8
		mouseClick (Timer)	36	1076	7.2%	1.86E+12	11.9
	Update	mouseClick (JButton)	1	51	7.8%	6.50 +9	64.1
		mouseClick (Timer)	23	206	12.2%	2.93E+12	44.44
	Delete	mouseClick (JButton)	1	29	0	1.15E+10	0
		mouseClick (Timer)	53	208	13.9%	1.93E+10	66.2

UI: User Interface **MI**: Method Invocation **AC**: Active Clone

This indicates that we are able to identify active clones through testing and that the reduced number of clones identified holds promise for focusing a software developer's clone inspection effort. It will be interesting to validate this assumption with real world maintenance evidence. A concern would be that ignoring other aspects of the codebase may result in unwanted faults. However, we assume that full regression testing would occur after any maintenance change and this would help to address any oversight.

Clones are considered harmful for a number of reasons, including: there is the potential for bug propagation during cloning, cloning increases the size of the codebase, and changes in one clone may require changes to other clones during maintenance. These issues increase the potential for a higher maintenance effort as well as may burden the software developer with an additional cognitive load. Dynamic analysis provides a way to aid in our comprehension of clones. Systems containing clones could be segmented based on testing to make the maintenance task more modular. As well, runtime information allows us to consider other clone

properties. Knowing which clones are the most CPU intensive during a particular set of system uses, could help identify places where optimizing one clone could be an important improvement to the system as a whole if we make similar changes to the clones in its clone class.

From Table 5.3 and Table 5.4 we see that for HSQLDB out of 9,825 methods in the system of which 1,323 are clones, we have only 35–40 active clones for each test. For JHotDraw, out of a 3,260 methods in the system of which 656 are clones, we have only 56–60 for each test, for RText, out of 3,698 methods in the system of which 189 are clones, we have only 11 active clones for the tests, for jEdit, out of 7,428 methods of which 419 are clones, we have 220 active clones for the test, and for uniCentaoPOS, out of 4,260 methodds of which 285 are clones, we have 8 active clones for the test. For HSQLDB, the active clones represent 492–545 lines code, out of 20,467 lines of clone code and a system that is 227,545 lines in size. For JHotDraw, the active clones represent 639–654 lines of code, out of 7,565 lines of clone code and a system size of 6,420. For RText, the active clones represent 265 lines of code, out of 3,019 lines of clone code and a 95,036 line system. For jEdit, the active clones represent 2304 lines of code, out of 7,320 lines of clone code and system line of code of 115246. For uniCentaoPOS, the active clones represent 55–141 lines of code, out of 4578 lines of code and a system size of 4,3032 With respect to number of files, HSQLDB has 513 files, 293 with method clones and between 22–25 with active clones for the uses represented by the use sets. The system JHotDraw has 309 files in the system, 183 with clones and between 26–28 files with active clones, and RText has 319 files, 81 with clones and 7 with active clones. The system jEdit 571 files in the system, 161 with clones and 44 with active clones while uniCentaoPOS contains 486 files, 174 with clones, and 4–15 files with active clones Also if we compare the percentage coverage of active clones, lines of code, files and clone classes (Table 5.5) we find that a fewer percentage of clones are active for each individual test.

We can see that although the systems have a high concentration of clones, relatively few of them are executed. Only those clones that are test specific are active and are shown to be used more often. For example if we consider the results in Table 5.5 then in the case of JHotDraw the lines of code for method clones is 13.4% (denoted by $\%L_{CLONE}$)) but only 1.2% (denoted by $\%L_{ACLONE}$) of code are containing active clones and likely requiring attention during maintenance. A similar picture can be seen for HSQLDB, RText, jEdit and uniCentaoPOS as well. On the other hand from Table 5.5 we can also associate the coverage of clones over files. For instance in HSQLDB 57% of files are associated with clones where only 17.3% on average are found in the trace (denoted by $\%F_{TRACE}$) while only 4.2% of file have active clones (denoted by $\%F_{ACLONE}$). For JHotDraw and RText the percentage of files in the trace denoted by $\%F_{TRACE}$ are 61.2% and 47.9% while the percentage of files containing active clones denoted by $\%F_{ACLONE}$ are 9.1% and 2.3% respectively. Similarly in case of jEdit and uniCentaoPOS $\%F_{TRACE}$ is 51.9% and 17.5% while $\%F_{ACLONE}$ is 7.4% and 3.1% respectively.

In summary, active clone detection has the following three characteristics if we just consider line, file and method metrics. First, tests can be used to identify active clones. Second, identifying active clone classes allows us to identify not only the clones that are active as a result of the test, but also (using static

Table 5.7: Frequently Executed Active Clones by System

System	Active Clone	Times Invoked	CPU Use (nanosec)	Clone Class	
				Id	Size
HSQLDB	readByte()	2509	1.02E+7	398	3
	getOrAddInteger()	421	2.48E+6	30	2
	writeShort()	43	1.07E+5	464	2
	readBoolean()	320	3.20E+4	398	3
	getInt()	5	4.09E+5	496	4
JHotDraw	basicGetBounds()	8	2.31E+6	207	6
	ensureSorted()	7	1.77E+6	186	2
	updateEnabledState()	6	8.05E+5	140	3
	fireToolDone()	1	9.56E+6	103	3
RText	getRTextScrollPaneAt()	33	5.10E+5	71	2
	createCheckBox()	6	1.51E+6	55	2
	windowGainedFocus()	6	1.04E+3	79	2
	windowLostFocus()	6	6.90E+3	79	2
	setSearchParameters()	1	1.29E+7	25	2
jEdit	getScreenLineStartOffset(int)	911	8.21+E9	196	2
	jj_3R_129()	234	3.12E+10	14	4
	jj_3R_216()	226	3.92E+9	31	4
	EqualityExpression()	90	9.71E+10	8	6
	jj_3R_37()	639	7.58E+10	129	6
uniCentaoPOS	readValues(int)	3	8.21+E9	67	2
	fireDataContentsChanged	1	2.7E+8	28	3
	activate()	1	8.14E+9	97	7
	fireDataIntervalAdded ()	1	7.57E+6	28	3
	getDialog(Component)	1	5.04E+8	69	3

analysis) the clones that are related to the active clones through clone classes and by seeing which clones are involved with which kind of system uses. It is important to consider all the related clones when performing maintenance to help ensure we consistently change them. We note that for the systems and uses we studied, the active clone classes are between 2 and 6 in size. Third, there is evidence that active clones are present in each test case we exercised. If we observe closely the statistics and comparison of active clones in terms of lines of code, files, and clone classes with active clones we can clearly see that active clones are very few in number involving less lines of code and less file coverage. For instance, in the case of HSQLDB a reduced number of lines of code, files and classes are found active per test. This may help guide a software developer in prioritizing which clones to look at by giving inspection priorities to active clones and their corresponding active clone classes instead of looking for every clone in the system.

RQ1: *To what extent are clones active during runtime?* Analysing our subject systems we see that clones are active even for simple tests and that with typical modest uses of a system that they are at least an order of magnitude fewer in number than the number of clones in a system. Active clones could be used to prioritize clone inspection during maintenance tasks by allowing us to focus on those clones related to the task at hand.

In addition to the previously discussed metrics, we can also analyze active clone runtime properties related to the traces, such as frequency of invocation, order of invocation, relation of invocation to typical system

uses, and resource utilization (i.e., CPU usage). Table 5.6 shows the active clone characteristic and the CPU usage of all the systems over each test case based on user interface (UI) events. The column *CPU time* denotes the total time consumed by events within the test and the column *%Active CPU Time* signifies the percent of CPU time consumed by the active clones executed by the corresponding event within the test.

From the Table 5.6 we see that concentration of active clones in feature implementation denoted by threadSignal is higher then GUI code with over 7% for HSQLDB for three of the tests and around 6.1% for JHotDraw which is less than GUI code, and 2.5% for RText which is higher than mouseClick but less than displayUpdate. Interestingly, if we compare the two mouse click events (buttons and mouse event) in JHotDraw we notice that there are a greater number of active clones associated with creating figures in the editor than clicking a button in the system. In case of jEdit threadSignal occupies 2.5% of active clones while for uniCentaoPOS more than 8% of the active clones are concentrated on the mouseClick (Timer) event with the exception of the test "Select" where all of the active clones are in mouseClick (JButton). From the Table 5.6 we can also observe that clones do execute and occupy a significant amount of CPU time. Thus, CPU time information of the active clones can also be an important parameter to consider during maintenance activity such as code optimization or determining inconsistent changes in clone. For instance, lets consider a class containing three clone fragments and all gets executed during a certain use. It is apparent that all the active clones should spent the same amount of time during execution. Therefore, any discrepancy in the timing value of the active methods could be an indication of certain method spending more time for execution. Possible reason could be a bug that have never been fixed here which made the methods inconsistent with other methods in the same class. In this way CPU time can be utilised to understand the behaviour of the methods and be used an an indicator of tracking abnormal behaviour of them in the clone class.

At the same time a segmented active clones can help developers identify which section of the system's code to inspect as a part of the maintenance task. Thus, there is a possibility that effort may be reduced by providing the developer with more detailed, fine grained data on active clones. For instance, if new features in the GUI are being added or modified as part of the maintenance task then the corresponding UI events and the associated active clones can help them in program comprehension. Code inspection may be reduced and the segmented active clones can further provide a way for effective clone management. As an example, we notice that for a number of uses, active clones are taking up a significant portion of the CPU time. For example, 33%–67% of the CPU time is involved with active clones for a small set of traces. If we look at Table 5.7 we can find that active clone methods gets executed in all the tests with higher number of frequencies and CPU time consumption in all the five systems. Interestingly, we can also observe in case of HSQLDB, RText and uniCentaoPOS that the fragments in the same clone class gets executed with a higher number of invocations. For instance, in HSQLDB methods readByte() and readBoolean() belong to the same clone class 398 and is invoked 2509 and 320 times which is a significant amount. For RText, methods windowGainedFocus() and windowLostFocus() belonging to the same clone class and gets executed six times

Table 5.8: Overview of Clone Genealogies

Domain	System	Versions
Text Editor	RText	1.3.1 - 2.0.7
Graphics	JHotDraw	7.0.6 - 7.5.1
Database Server	HSQLDB	2.2.0 - 2.2.10
Point of Sale	UnicentaoPOS	2.53 - 3.02
Text Editor	jEdit	4.3.2-5.00

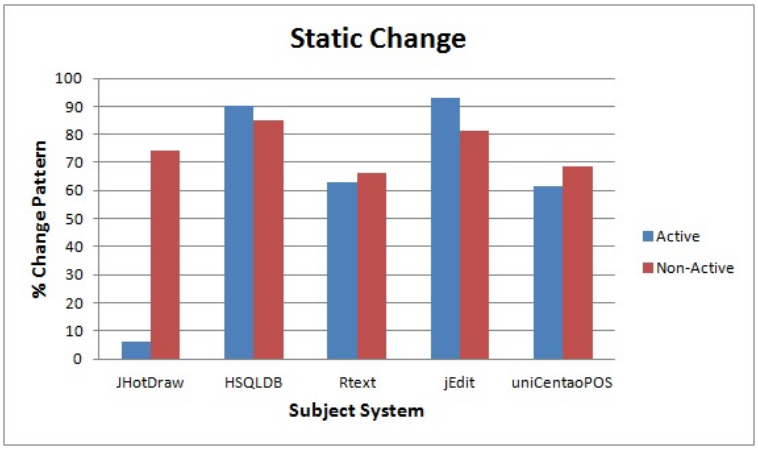

Figure 5.1: Static Change: Active Versus Non-Active

each while for uniCentaoPOS methods fireDataContentsChanged() and fireDataIntervalAdded() are invoked once.

RQ2: *How active are the active clones?* The research question RQ1 addressed the dynamic behaviour of the active clones involved during the runtime of the systems. It signified that clones do exercise when a system is used and each feature contains active clones. Since maintenance of a system is a continuous process, changes in the code is done to meet customer requirements, fix bugs or optimize to make the system more efficient. Therefore software evolves with time and during which changes are made to the systems. In our study we are interested to look at how the active clones participate in the maintenance phase when the developers modify the source code. More specifically we wanted to study how the active clones evolve. We have adapted the genealogy extractor, called gCAD [83] to extract the genealogy of the systems over a number of versions as shown in Table 5.8 to detect the change patterns of the systems. We then post processed the genealogy of the systems to map the active clones and identified their patterns. In our study we have considered only three types of change patterns : a) static b) consistent and c) inconsistent change patterns. We then compared the activeness of clones against the active and the non-active code clones. Figures 5.1, 5.2, and 5.3 show the change patterns of the active clones and non-active clones for each systems.

From Figure 5.1 we can observe that both the non-active clones and the active clones change and more than 60% of the clones play a significant role during the change. In case of HSQLDB, RText, jEdit and uniCentaoPOS we can see a similar pattern that the difference between the active and non-active code clones

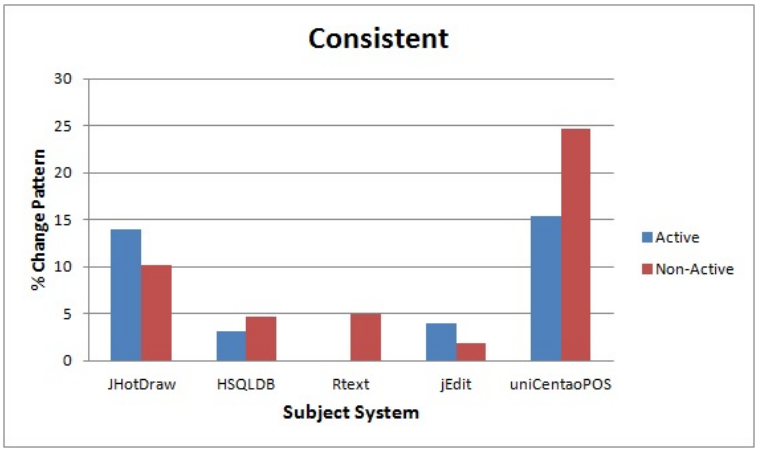

Figure 5.2: Consistent Change: Active Versus Non-Active

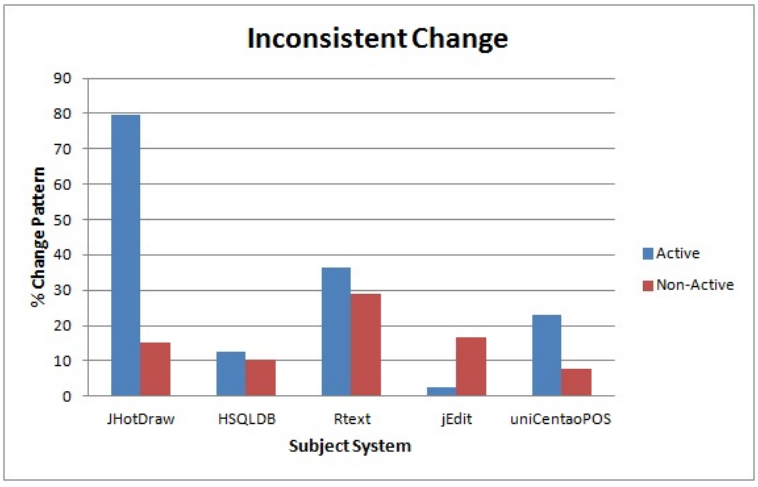

Figure 5.3: Inconsistent Change: Active Versus Non-Active

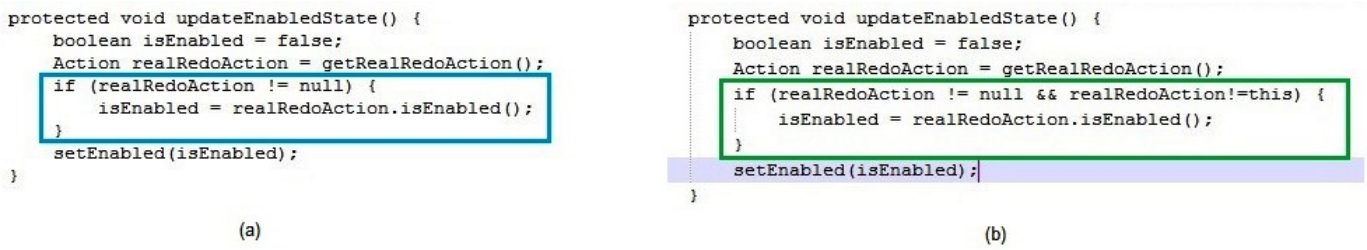

Figure 5.4: Code Snippets from JHotDraw Showing Clone Fragments in Two Versions: (a) Version 7.0.6 and (b) Version 7.4.1

on an average is less than 10% with an exception of JHotDraw where the difference is around 40% and the activeness of the non-code clones surpasses the active clones. Therefore, we can conclude that active clones are significantly active in terms of change as the non-active clones during the static change pattern.

On the other hand, if we look closely at Figure 5.2 for the consistent change, we can see that for the systems JHotDraw, HSQLDB, jEdit and uniCentaoPOS both the active and the non-active clones change at the same time and the difference in activeness between them is less than 6% with an exception of RText where no active clones participate in the change pattern. Therefore, active clones do play a vital role and have significant impact on the maintenance activity.

Though practice of code cloning speeds up the development process, it also poses threat of bug propagation due to inconsistently changing the clone fragments. If a clone fragment is changed for bug fixing and at the same time the corresponding fragments are left unchanged then the bugs will sustain and propagate further. This may also introduce new bugs later during the use of the software system. Therefore, we also looked into the comparison of the behaviour of the active and the non-active code clones in terms of inconsistent changes. If we observe Figure 5.3 we can find that both the active and the non-active clones behave in a similar fashion such that each system contains clones that are changed inconsistently. Interestingly, we can also see that active clones are more active than the non-active clones and surpasses the non-active ones. In case of JHotDraw, HSQLDB, RText and uniCentaoPOS the active clones have changed more inconsistently than the non-active clones. In case of JHotDraw almost 80% of the active clones are changed inconsistently compared to non-active clones which is around 15%. Similarly for HSQLDB, RText and uniCentaoPOS the percentage of inconsistent changes of active and non-active clones are around 15%, 35%, 22% and 10%, 29% and 8% respectively. However in case of jEdit even though the non-active clones have changed more inconsistently than the active clones we still get some active clones that have changed inconsistently.

To get more insight we randomly picked active clone classes and analysed the source code of the active clones manually. Our intent was to look into the purpose for which active clones have been changed. The following Figures 5.4 and 5.5 shows the active clone code snippets from JHotDraw that have been modified over to the next versions.

Figure 5.4 (a) shows a comparison of the snippets between version 7.0.6 and 7.4.1. In version 7.0.6 the statement *isEnabled = realRedoAction.isEnabled())* has been guarded by a second condition. This is

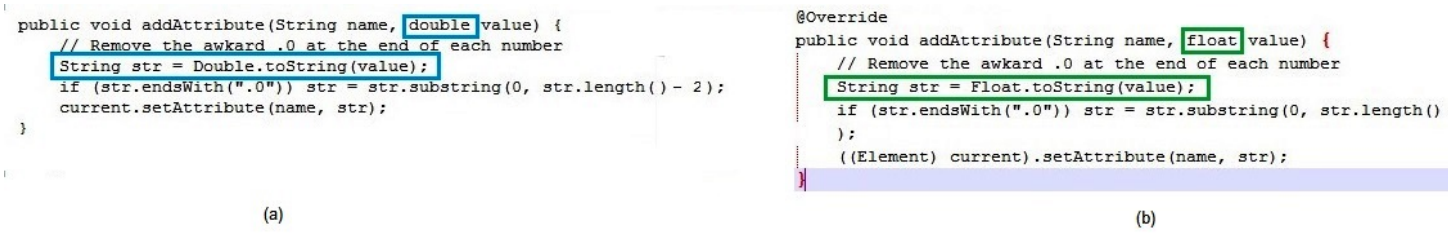

Figure 5.5: Code Snippets from JHotDraw Showing Clone Fragments in Two Versions: (a) Version 7.0.6 and (b) Version 7.5.1

Table 5.9: Number of Genealogies of Active and Non-Acitve Clones by Type

System	Type	Active Clones			Non-Active Clones		
		Static	CC	IC	Static	CC	IC
JHotDraw	Type-1	7	2	6	60	43	60
	Type-2	1	2	9	53	47	64
	Type-3	3	6	31	158	66	193
HSQLDB	Type-1	2	0	0	76	3	5
	Type-2	2	1	0	75	6	6
	Type-3	25	0	2	283	15	42
RText	Type-1	0	0	1	9	3	2
	Type-2	3	0	0	37	2	6
	Type-3	4	0	3	75	4	45
jEdit	Type-1	1	0	0	3	0	2
	Type-2	25	0	0	41	2	3
	Type-3	45	3	2	82	1	21
uniCentaoPOS	Type-1	0	0	0	15	2	1
	Type-2	4	0	0	30	12	1
	Type-3	6	2	3	76	30	14

a possible indication of bug fixation. Moreover, we also noted that during the evolution process the clone class has been changed inconsistently with new fragments being added and deleted to the clone classes. On the other hand, Figure 5.5(b) gives us a different insight about the reason for the modification of the code snippets. If we take a close look into both the codes of versions 7.0.6 and 7.5.1 we can see that parameter type from double has been changed to float in the method *addAttribute()* from version 7.0.6 to version 7.5.1. The variable double is a 64 bit data type whereas float is a 32 bit data type which is half the storage consumption of the double.

RQ3: *Does active clone identification support software maintenance activities?* From our study we see that awareness of active clones and their runtime behaviour may help support maintenance by allowing developer's to focus on similar code fragments whose improvement could benefit the use of the system as a whole and on clone classes that are involved in system uses related to the task at hand. This is well supported by the research questions RQ1 and RQ2. The dynamic properties of active clones such as times of invocations, CPU time, and active clone genealogies can be used to prioritize clone inspection during maintenance activities. Clones that are invoked a greater number of times and that occupy higher percentage of CPU time can be put at the highest priority in the list of clone inspection during clone management. Moreover, active clone

genealogies can also assist the maintenance engineers to formulate the clone inspection process by selecting the active clone classes based on their change patterns. Since a clone class contains a number of similar fragments and the active clone classes being a part of genealogies, both can indicate the engineers to consider all the fragments in that clone class during modifications. This will also avoid clone fragments that are left with inconsistent change and eventually prevent further bug propagation.

Previous study [84] on the change patterns on the genealogies of clones show that Type-3 clones are more inconsistent than the Type-1 and Type-2 clones which require more careful attention during maintenance. With regards to this, we also tried to conduct a similar study based on our framework to reveal the change pattern of the active clone types. Table 5.9 shows the comparison of activeness of Type-1, Type-2 and Type-3 clones between the active and non-active clones. If we observe closely, we can see that the Type-3 clones have a greater tendency to change when compared to Type-1 and Type-2 clones. In all the change patterns, the Type-3 active clones surpasses that of Type-1 and Type-2 clones. More interestingly, on an average 31.7% of Type-3 active clones change inconsistently while Type-1 changes slightly less than 29%. Similarly, for consistent change pattern 4.2% of active clones are Type-3 with no change for Type-1 and Type-2. On an average for static change, 60.4% of Type-3 clones are static in nature followed by Type-1 which is 49.2% with Type-2 clones standing at 74.6%. We also noticed that Type-3 clones are significantly less stable than the Type-2 clones with an exception of Type-1 clones. This is because of the absence of Type-1 active clone fragments in RText and uniCentaoPOS. In case of non-active clones for all the three change patterns we can see that Type-3 clones possess inconsistent changes with over 50% followed by Type-2 and Type-1. Therefore, the study conducted in the activeness of active clones based on types can also add benefits to software maintenance, especially when clones need to be managed. We can conclude that further enhancement in clone management activity can be done by prioritizing the clone inspection based on types. From the study (Table 5.9) we can infer that clone types that are less stable can be looked at first than the other ones that are more stable. In other words, we can say that active clones that have a higher tendency to change inconsistently can be given higher attention than the consistent ones so that they can be managed carefully.

5.4 Summary

There are a number of threats to the validity of our analysis. Execution traces are based on the tests we provided. Additional testing would reasonably result in additional methods being executed with the possibility of additional active clone fragments being identified and thus the reduction in clones to be considered may not be as significant. However, our intention of this study is to only focus on test specific clones during the maintenance of that related functionality, and it is obvious that there will be a reduction of active clone fragments (hence the reduction of active clone classes) for maintenance as evident by our results.

In our study, we did not use real test suites of the systems to address any specific maintenance tasks.

As a result, all the tests were synthetic in nature which impacted the relevance of the collected execution traces. Again, this does not have any impacts on our findings as it is obvious that there will be less number of active clones for a test case compared to all the clones of the system. For the same reason, we did not feel the importance of making a comprehensive test suite that can exercise the entire subject system. In the case where the entire system is exercised by a test suite, all the clones of that system will be active clones and thus all the clones are subject to maintenance, which is in contrary to the motivation of this work.

We studied a small set of systems and thus it is impossible to generalize the findings. However, we carefully selected the systems of diverse varieties and sizes and experienced similar characteristics in favour of our claim. Furthermore, it is of course obvious that a subsystem (i.e., a part of the system captured by a test suite) will have less number of clone fragments than the entire system, and essentially will help reduce the maintenance effort. Our framework is designed to report active clone fragments of the subset of the system's features. If the active clone fragments of the whole system is desired then the tool will report all the clones of the system as active clones, and thus prioritization of them followed by reduction in management effort does not apply.

Another threat to the validity of the findings (in particular the precision and recall of finding active clone fragments) is that it depends on a third party static clone detection tool. Depending on the tool, the types and granularity of clones, and the precision and recall of finding active clones might significantly vary. However, in our approach, we have used NiCad, which works both for method and block levels of clones, and which has been shown highly accurate in terms of both precision and recall both for method and block levels of clone detection. In order to gain further confidence on the findings, we have applied a semi-automated search based approach for a number of active clones. We randomly selected a few active clone fragments and then for each of them we searched for their corresponding similar fragments in the source code using an IDE based near-miss clone detection tool developed in our lab [105]. For all the cases, we experienced similar findings as of NiCad. We also conducted the same procedure for a few other methods that were found in the traces but were not part of the clones. We did not find any other fragments similar to them in the system, which essentially again is consistent with NiCad's output.

Chapter 6

Conclusion and Future Work

Software maintenance is an important aspect in software engineering which ensures sustainability of efficient, reliable and right product for customers. It is an incremental process which initiates after a product's release. Meeting a customer requirement is one of the prime objectives of maintenance activity. Thus maintaining a system consumes a considerable amount of time. The process even becomes complex during the course of its usage as more and more modifications are brought into it making the system large in size. Therefore, this impacts program comprehension as well such that poor understanding about a system's implementation, architecture and design may cause the maintenance difficulty or impossible in many cases.

Understanding the features of a system by analysing the method calls collected as execution traces can help the developer to gain insight of how a feature is implemented. Execution traces contain a sequence of method calls that represents the underlying implementation of features. Knowing only the method calls in textual format will not provide any understanding about a system's implementation. Therefore, mining the execution traces is one of the necessary steps. Revealing the context (objective of implementation of methods) can add benefit to the developers by making the traces more meaningful. If the method calls can be segmented based on system use, test and event then the features can be located easily. Our technique contains simple yet efficient and when implemented as a tool makes the analysis of trace easier and comprehensive. We have defined frameworks for testing, mining, and visualization. We have used Aspect Oriented Programming (AOP) approach to instrument code and then a static one to mine important information from the source code.

Locating features in a system is one of the challenging tasks for program comprehension. The better a feature can be located the more precise and perfect would the maintenance be. To make the analysis more easy, simple and yet effective we have performed mining of traces both in terms of quantification and clustering. The more modular and informative a trace would be the easier would be the comprehension. To address this issue we have categorised method calls based on system use, test and events to determine their contexts. We have also proposed a taxonomy of metrics and enhanced the understanding of the features by applying them to quantify their behaviour. The comprehension process is further accompanied by using the display of screen shots in the form of image. This solves the problem of mapping segmented traces to interfaces. We have also used an annotator to segment execution traces during the testing so that the beginning and the end of tests can be easily identified. We also mined information for similar code fragments

called clones to determine the contexts in which they are used. Finally, to support the clone management we further mined the traces to determine the behavioural aspect of the clones using a set of active clone metrics.

In chapter 3, we have proposed a testing framework, an analysis framework, a trace mining framework, a taxonomy of metrics and a trace mining data organization. The testing framework is used to connect the user activities defined in use sets and tests to the source code. It accommodates tests corresponding to a user interacting with the system. The analysis framework describes the overall strategy of our study of how data is collected and the process of the execution trace mining. Our framework incorporates both the static and the dynamic approaches. The static approach is used to mine source code information and the dynamic approach is used to collect run time data. The trace mining framework is used to mine the execution trace data to identify features, clustering of methods based on API and clones, and the metrics calculation to determine the behavioural aspects of the features.

In Chapter 4, we have described the prototype of TrAM. We created a trace mining tool with interactive interface that complements the method call based feature analysis of execution traces. The tool allows the developers to navigate through a tree-like structure of the execution traces, identify features and understand their context. It contains panels such as main panel for the representation of traces, an image thumbnail panel for displaying user interfaces, a metric panel to display the metric values based on system uses, tests, events, methods and groups. The thumbnail panel can be zoomed out and the code viewer panel displays the real implemented source code associated with the snapshot. To differentiate between the internal and external methods, the tool incorporates a filtering approach that allows the developers to analyse features based on choice of methods. Three radio buttons internal, external or both selection choices can let the developers to switch among them as desired. Grouping of methods based on API and active clones is supported by the categorization panel. The developers can easily analyse the existence of groups of methods and active clone class containing fragments in the execution trace.

In Chapter 5, we have also conducted an empirical study on the execution traces using our tool to examine the behaviour of active clones. We have answered three research questions by conducting experiments on five subject systems JHotDraw, HSQLDB, RText, jEdit, and UniCentaoPOS of varying size and types. We found that traces do contain active clones and in every test there is involvements of active clones. We also studied active clones in terms of their activeness using the active clone genealogy and compared their change patterns with the non-active clones. We found that active clones are highly active than the non-active clones and change more inconsistently. Moreover, we also found that active clones are less stable than the non-active clones. On further analysis we also found that Type 3 active clones are more active than the Type 2 and the Type 1 clones. Therefore, active clones can be used as a means for clone management such as prioritizing clone inspection. The active clones can also help the maintenance engineers to understand what portions of trace are occupied by active clones and the how distributed the active clones are over the system. Although we have used systems of various size and types, analysis on more system would provide precise result. Similarly, the experiment is sensitive to test and its coverage where real tests would have provided

more precise results. We presented an approach for assisting a software developer in the maintenance of systems containing clones by focusing on those clones that are involved in tests related to the maintenance task.

One of the major part of trace mining in TrAM is active clone mining. Right now it is dependent on a single clone detector NiCad having a its own file format of clone representation. In future, we plan to mine active clone information from the result file of other clone detector having different file formats. The traces that are shown in the trace screen are plain texts of method calls and their call graph. To make it more easy, simple and readable to the user we plan to collapse the trace based on consecutive repeated patterns. For instance, a certain portion of code may contain in a loop and at certain point it executes a number of times with identical emerging patterns. Such pattens can be reduced by clustering them all into one group and collapsed as one. This will reduce the length and size of the trace. To do so, we plan to make use of any kind of similarity pattern to detect similar chunks of trace. The call graph represented by TrAM is a caller-callee relationship. However, it can be enhanced and made more informative by adding the information of caller-callee relationship in terms of inheritance in case of API.

The other part of the mining of TrAM is the quantification of trace in terms of metrics. To make them more presentable and understandable, graphical approaches with color schemes can be implemented. From the user study we have also found that 36% of the participants had difficulty in reading information of traces due to the complex organization of data. We, therefore, also plan to make the organization of the tool more simple and yet more presentable to the users. Moreover, the user study also revealed that 43% of the participants had other comments about the tool's design and appearance. The majority of them expressed their interests on having a help option that will provide them with a quick guideline on how to use the tool and what the features represent. Few of them wanted more categorization based on groups of internal methods such as class based consumption of the internal methods. Thus, our future work will be concentrated on the issues raised by the participants and plan to resolve them by adding those new functionalities. Besides categorizing trace based on external methods and active clone classes, we will also incorporate categorization by method return types, and access modifier. For instance, a method may have a return type of int, float, double, object and so on and categorizing the methods based on these types would be interesting to analyse. Finally, we will incorporate more dynamic information of the method's resource consumption such as amount of memory allocated by the internal and external methods.

One of the main applications of the tool is to assist the new developers to understand the functionalities of a system. In industry where a lot of commercial systems are developed must go through the maintenance phases that would require changes to be made in the source code. Therefore, a system evolves continuously to meet the customer requirements. In such cases, execution traces can be collected for all the functionalities implemented by the system. Therefore, whenever a new developer is on its way to modify a certain functionality then at first they can use TrAM to understand the implementation of the features which will save a lot of maintenance time dedicated for understanding.

References

[1] K. Arnold. Programmers are People, Too. ACM Queue, volume 3, issue 5, pages 54–59, 2005.

[2] F. Asadi, M. Di Penta, G. Antoniol, and Y. Gueheneuc. A Heuristic-based Approach to Identify Concepts in Execution Traces. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering* (CSMR '10), pages 31-40, 2010.

[3] M. Asaduzzaman, C. K. Roy, and K. Schneider. VisCad: Flexible Code Clone Analysis Support For NiCad. In *Proceedings of the International Workshop on Software Clone* (IWSC '11), pages 77–78, 2011.

[4] T. Bakota, R. Ferenc, and T. Gyimothy. Clone smells in software evolution. In *Proceedings of the International Conference on Software Maintenance* (ICSM '07), pages 24–33, 2007.

[5] T. Ball. The Concept of Dynamic Analysis. In *Proceedings of 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of software engineering* (ESEC/FSE '99), volume 1687, issue 1999, pages 216-234, 1999.

[6] T. T. Bartolomei, K. Czarnecki, R. Lammel, and T. van der Storm. Study of an API migration for two XML APIs. In *Proceedings of 2nd International Conference on Software Language Engineering* (SLE '09), pages 42–61, 2009.

[7] V. R. Basili. Evolving and packaging reading technologies. *Journal of System Software* volume 38, issue 1, pages 3-12, 1997.

[8] J. Bloch. How to Design a Good API and Why it Matters. In *Proceedings of 21st International Conference on Object-oriented Programming Systems, Languages, and Applications* (OOPSLA '06), pages 506–507, 2006.

[9] C. Bore and S. Bore. Profiling Software API Usability for Consumer Electronics. In *Proceedings of International Conference on Consumer Electronics* (ICCE '05), pages 155–156, 2005.

[10] G. Canfora and A. Cimitile. Software maintenance. In *Handbook of Software Engineering and Knowledge Engineering*, 2002.

[11] S. Clarke and C. Becker. Using the Cognitive Dimensions Framework to Evaluate the Usability of a Class Library. In *Proceedings of Joint Conference of EASE and PPIG*, pages 359–366, 2003.

[12] J. R. Cordy and C. K. Roy. Tuning Research Tools for Scalability and Performance: The NICAD Experience. *Science of Computer Programming* 79,1 (January 2014), pp. 158-171.

[13] J. R. Cordy and C.K. Roy. The NiCad Clone Detector. In *Proceedings of the Tool Demo Track of the 19th International Conference on Program Comprehension* (ICPC '11), pages 219–220, 2011.

[14] B. Cornelissen, A. Zaidman, A. Deursen, L. Moonen, and R. Koschke. A Systematic Survey of Program Comprehension through Dynamic Analysis. In *IEEE Transaction on Software Engineering*, volume 35 issue 5, pages 684–699, 2009.

[15] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen and A. van Deursen, and J. J. van Wijk. Execution Trace Analysis Through Massive Sequence and Circular Bundle Views. *The Journal of Systems and Software* 81 (2008), pages 2252-2268, 2008.

[16] U. Dekel and J. D. Herbsleb. Improving API Documentation Usability with Knowledge Pushing. In *Proceedings of International Conference on Software Engineering* (ICSE '09), pages 16–24, 2009.

[17] T. Eisenbarth, R. Koshke, and D. Simon. Aiding Program Comprehension by Static and Dynamic Feature Analysis. In *Proceedings of the International Conference on Software Maintenance* (ICSM '01), pages 602–611, 2001.

[18] B. Ellis, J. Stylos, and B. Myers. The factory pattern in API design: A Usability Evaluation. In *Proceedings of the 29th International Conference on Software Engineering* (ICSE '07), pages 302–312, 2007.

[19] M. D. Ernst. Static and Dynamic Analysis: Synergy and Duality. In *Proceedings of ICSE Workshop on Dynamic Analysis*, pages 25-28, 2003.

[20] R. E. Fairley. Tutorial: Static Analysis and Dynamic Testing of Computer Software. *IEEE Journals and Magazines*, pages 14-23, 1978.

[21] M. Feilkas and D. Ratiu. Ensuring Well-Behaved Usage of APIs through Syntactic Constraints. In *Proceedings of the 16th IEEE International Conference on Program Comprehension* (ICPC '08), pages 248–253, 2008.

[22] M. Fischer, J. Oberleitner, H. Gall, and T. Gschwind. System Evolution Tracking Through Execution Trace Analysis. In *Proceedings of the 13th International Workshop on Program Comprehension* (IWPC '05), pages 237–246, 2005.

[23] R. Geiger, B. Fluri, H. C. Gall, and M. Pinzger. Relation of code clones and change couplings. In *Proceedings of the 9th International Conference of Fundamental Approaches to Software Engineering* (FASE '06), pages 411–425, 2006.

[24] J. D. Gradecki and N. Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java.* John Wiley and Sons, Pages 456, 2003.

[25] T. Gschwind, J. Oberleitner, and M. Pinzger. Using Run-Time Data for Program Comprehension. In *Proceedings of the 11th International Workshop on Program Comprehension* (IWPC '03), pages 245–250, 2003.

[26] A. Hamou-Lhadj and T. Lethbridge. Measuring Various Properties of Execution Traces to Help Build Better Trace Analysis Tools. In *Proceedings of the 10th International Conference on Engineering of Complex Computer Systems* (ICECCS '05), pages 559–568, 2005.

[27] A. Hamou-Lhadj and T. Lethbridge. Compression Techniques to Simplify the Analysis of Large Execution Traces. In *Proceedings of the 10th International Workshop on Program Comprehension* (IWPC '02), pages 159–168, 2002.

[28] A. Hamou-Lhadj. *Techniques to Simplify the Analysis of Execution Traces for Program Comprehension.* PhD Thesis, http://www.site.uottawa.ca/∼tcl/gradtheses/ahamou/AbdelwahabHamouLhadjPhDThesis.pdf, University of Ottawa, pages 171, 2005.

[29] A. Hamou-Lhadj and T. C. Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research* (CASCON '04), pages 42-55, 2004.

[30] A. Hamou-Lhadj, T. Lethbridge, and L. Fu. Challenges and Requirements for an Effective Trace Exploration Tool. In *Proceedings of International workshop on Program Comprehension* (IWPC '04), pages 70–78, 2004.

[31] M. J. Harrold. Testing: a Roadmap. In *Proceedings of the Conference on The Future of Software Engineering* (Special Volume published in conjunction with ICSE '00), pages 61–72, 2000.

[32] J. Henkel and A. Diwan. CatchUp!: Capturing and Replaying Refactorings to Support API Evolution. In *Proceedings of the 27th International Conference on Software Engineering* (ICSE '05), pages 274–283, 2005.

[33] M. Henning. API Design Matters. ACM Queue, volume 5, issue 4, pages 24–36, 2007.

[34] J. N. Herder. Aspect-Oriented Programming with AspectJ. Vrije Universiteit, Amsterdam, 2002.

[35] R. Holmes, R. Walker, and G. Murphy. Approximate Structural Context Matching: An Approach to Recommend Relevant Examples. In *IEEE Transactions on Software Engineering*, volume 32, issue 12, pages 952–970, 2006.

[36] P. Jalote, V. Vangala, T. Singh, and P. Jain. Program Partitioning - A Framework for Combining Static and Dynamic Analysis. In *Proceedings of the International Workshop on Dynamic systems Analysis* (WODA '06), pages 11– 16, 2006.

[37] D. Jerding and S. Rugaber. Using Visualisation for Architecture Localization and Extraction. In *Proceedings of the 4th Working Conference on Reverse Engineering* (WCRE '97), pages 56–65, 1997.

[38] D. Jerding, J. Stasko, and T. Ball. Visualizing Interactions in Program Executions, In *Proceedings of the International Conference on Software Engineering* (ICSE '97), pages 360–370, 1997.

[39] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering* (ICSE '09), pages 485–495, 2009.

[40] C. Kapser and M. W. Godfrey. Cloning considered harmful. In *Proceedings of the 13th Working Conference on Reverse Engineering* (WCRE '06), pages 19–28, 2006.

[41] D. Kawrykow and M. P. Robillard. Detecting Inefficient API Usage. In *Proceedings of 31st International Conference on Software Engineering - Companion* (ICSE-Companion '09), pages 183–186, 2009.

[42] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming* (ECOOP '01), pages 327–353, 2001.

[43] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An Empirical Study of Code Clone Genealogies. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT international Symposium on Foundations of Software Engineering* (ESEC-FSE-13 '05), pages 187-196, 2005.

[44] K. Koskimies and H. Mossenbock. Scene: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs. In *Proceedings of the 18th International Conference on Software Engineering* (ICSE '96), pages 366–375, 1996.

[45] K. Koskimies, T. Mannisto, T. Systa, and J. Tuomi. SCED: A Tool for Dynamic Modeling of Object Systems. University of Tampere, Department of Computer Science, Report A-1996-4, 1996.

[46] R. Laddad. AspectJ in Action. Second Edition, pages 568, 2009.

[47] D. Lafferty and V. Cahill. Language-Independent Aspect-Oriented Programming. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA '03), pages 1–12, 2003.

[48] D. B Lange and Y. Nakamura. Object-Oriented Program Tracing and Visualization, *IEEE Computer Society*, volume 30, issue 5, pages 63–70, 1997.

[49] T.D. LaToza, G. Venolia, and R. DeLine. Maintaining Mental Models: A Study of Developer Work Habits. In *Proceedings of the 28th International Conference on Software Engineering* (ICSE '06), pages 492–501, 2006.

[50] S. Liang and D. Viswanathan. Comprehensive Profiling Support in the Java Virtual Machine. In *Proceedings of the USENIX Conference On Object Oriented Technologies and Systems* (COOTS '99), pages 229-240, 1999.

[51] D. Lo, S. Khoo, and C. Liu. Mining Temporal Rules from Program Execution Traces. In *Proceedings of the 3rd International Workshop on Program Comprehension through Dynamic Analysis* (PCODA '07), pages 24–28, 2007.

[52] A. Lozano and M. Wermelinger. Assessing the Effect of Clones on Changeability. In *Proceedings of the International Conference on Software Maintenance* (ICSM '08), pages 227–236, 2008.

[53] L. Luo. Software testing techniques. *Institute for Software Research International Carnegie Mellon University Pittsburgh*, PA, 15232:1-19, 2001.

[54] H. Ma, R. Amor, and E. Tempero. Usage Patterns of Java Standard API. In *Proceedings of 13th Asia Pacific Software Engineering Conference* (APSEC '06), pages 342–352, 2006.

[55] S. McLellan, A. Roesler, J. Tempest, and C. Spinuzzi. Building more Usable APIs. In *IEEE Software*, volume 15, issue 3, pages 78–86, 1998.

[56] M. Mondal, C. K. Roy, and K. A. Schneider. Insight into a Method Co-change Pattern to Identify Highly Coupled Methods: An Empirical Study. In *Proceedings of the 21st IEEE International Conference on Program Comprehension* (ICPC '13), pages 103–112, 2013.

[57] M. Mondal, C. K. Roy, and K. A. Schneider. Connectivity of Co-changed Method Groups: A Case Study on Open Source Systems. In *Proceedings of the 2012 Conference of the Centre for Advanced Studies on Collaborative Research* (CASCON '12), pages 205–219, 2012.

[58] M. Mondal, C. K. Roy, and K. A. Schneider. An Empirical Study on Clone Stability. *ACM SIGAPP Applied Computing Review* (ACR), volume 12, issue 3, pages 20–36.

[59] M. Mondal, C.K. Roy, and K.A. Schneider. Dispersion of Changes in Cloned and Non-cloned Code. In *Proceedings of the ICSE 6th International Workshop on Software Clones* (IWSC '12), pages 29–35, 2012.

[60] M. Mondal, C. K. Roy, M. S. Rahman, R. K. Saha, J. Krinke, and K.A. Schneider. Comparative Stability of Cloned and Non-cloned Code: An Empirical Study. In *Proceedings of the Software Engineering Track of the 27th ACM Symposium on Applied Computing* (ACM SAC '12), pages 1227-1234, 2012.

[61] J. S. Paudel. *Aspect Structure of Compilers.* PhD Thesis, http://ecommons.usask.ca/bitstream/handle/10388/etd-08272009-164252/ThesisJeeva.pdf, University of Saskatchewan, pages 135, 2009.

[62] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualizing the Execution of Java Programs. *Revised Lectures on Software Visualization*, pages 151–162, 2001.

[63] W. De Pauw, D. Lorenz J. Vlissides, and M. Wegman. Execution Patterns in Object-Oriented Visualization. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems* (COOTS '98), pages 219–234, 1998.

[64] D. J. Pearce, M. Webster, R. Berry, and P. H.J. Kelly. Profiling with AspectJ. *Software: Practice and Experience*, volume 37, issue 7, pages 747–777, 2007.

[65] D. J. Pearce, P. H. J. Kelly, T. Field, and U. Harder. Gilk: A Dynamic Instrumentation Tool for the Linux Kernel. In *Proceedings of the International TOOLS Conference*, pages 220-226, 2002.

[66] H. Pirzadeh, A. Agarwal, and A. Hamou-Lhadj. An Approach for Detecting Execution Phases of a System for the Purpose of Program Comprehension. In *Proceedings of 8th ACIS International Conference on Software Engineering Research, Management and Applications* (SERA '10), pages 207-214, 2010.

[67] H. Pirzadeh and A. Hamou-Lhadj. A Novel Approach Based on Gestalt Psychology for Abstracting the Content of Large Execution Traces for Program Comprehension. In *Proceedings of 16th International Conference on Engineering of Complex Computer Systems* (ICECCS '11), pages 221–230, 2011.

[68] H. Pirzadeh, A. Hamou-Lhadj, and M. Shah. Exploiting Text Mining Techniques in the Analysis of Execution Traces. In *Proceedings of 27th IEEE International Conference on Software Maintenance* (ICSM '11), pages 223–232, 2011.

[69] M. S. Rahman, A. Aryani, C. K. Roy, and F. Perin. On the Relationships between Domain-Based Coupling and Code Clones: An Exploratory Study. In *Proceedings of the New Ideas and Emerging Results Track of the 35th International Conference on Software Engineering* (ICSE '13), pages 1265–1268, 2013.

[70] T. Richner and S. Ducasse. Using Dynamic Information for the Iterative Recovery of Collaborations and Roles. In *Proceedings of the 18th International Conference on Software Maintenance* (ICSM '02), pages 34–43, 2002.

[71] M. Rieger, S. Ducasse, and M. Lanza. Insights into System-wide Code Duplication. In *Proceedings of the 11th Working Conference on Reverse Engineering* (WCRE '04), pages 100–109, 2004.

[72] C. K. Roy and J. R. Cordy, 2010. Near-miss Function Clones in Open Source Software: An Empirical Study. *Journal of Software: Evolution and Process*, volume 22 issue 3, pages 165–189.

[73] C. K. Roy and J. R. Cordy, 2008. An Empirical Evaluation of Function Clones in Open Source Software. In *Proceedings of the 15th Working Conference on Reverse Engineering* (WCRE '08), pages 81–90, 2008.

[74] C. K. Roy and J. R. Cordy. A survey on software clone detection research. *School of Computing Technical Report* 2007-541, Queens University, Canada, pages 115, 2007.

[75] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming*, pages 470–495, 2009.

[76] C. K. Roy and J. R. Cordy. A Mutation/Injection-based Automatic Framework for Evaluating Code Clone Detection Tools. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops* (ICSTW '09), pages 157–166, 2009.

[77] C. K. Roy and J. R. Cordy. NiCad: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *Proceedings of the 16th International Conference on Program Comprehension* (ICPC '08), pages 172–181, 2008.

[78] C. K. Roy. Detection and Analysis of Near-Miss Software Clones. In *Proceedings of the Doctoral Symposium Track of the 25th IEEE International Conference on Software Maintenance* (ICSM '09), pages 447–450, 2009.

[79] C. K. Roy and J.R. Cordy. Towards a Mutation-Based Automatic Framework for Evaluating Code Clone Detection Tools. In *Proceedings of the Poster Paper Track of the Canadian Conference on Computer Science and Software Engineering* (C3S2E '08), pages 137–140, 2008.

[80] C. K. Roy, M.G. Uddin, B. Roy, and T. R. Dean. Evaluating Aspect Mining Techniques: A Case Study. In *Proceedings of the 15th IEEE International Conference on Program Comprehension* (ICPC '07), pages 167–176, 2007.

[81] H. Safyallah and K. Sartipi. Dynamic Analysis of Software Systems using Execution Pattern Mining. In *Proceedings of the 14th International Conference on Program Comprehension* (ICPC '06), pages 84–88, 2006.

[82] R. K. Saha, M. Asaduzzaman, M. F. Zibran, C.K. Roy, and K. A. Schneider. Evaluating Code Clone Genealogies at Release level: An Empirical Study. In *Proceedings of the 10th IEEE International Conference on Source Code Analysis and Manipulation* (SCAM '10), pages 87–96, 2010.

[83] R. K. Saha, C. K. Roy, and K. A. Schneider. An Automatic Framework for Extracting and Classifying Near-Miss Clone Genealogies. In *Proceedings of the 27th IEEE International Conference on Software Maintenance* (ICSM '11), pages 293–302, 2011.

[84] R. K. Saha, C. K. Roy, K. A. Schneider, and D. E. Perry. Understanding the Evolution of Type-3 Clones: An Exploratory Study. In *Proceedings of the 10th Working Conference on Mining Software Repositories* (MSR '13), pages 139–148, 2013.

[85] M. Salah, S. Macoridis, G. Antoniol, and M. Di Penta. Scenario-Driven Dynamic Analysis for Comprehending Large Software Systems. In *Proceedings of 10th European Conference on Software Maintenance and Reengineering* (CSMR '06).

[86] J. Stylos and S. Clarke. Usability Implications of Requiring Parameters in Objects' Constructors. In *Proceedings of the 29th International Conference on Software Engineering* (ICSE '07), pages 529–539, 2007.

[87] J. Stylos, B. Graf, D. K. Busse, C. Ziegler, R. Ehret, and J. Karstens. A Case Study of API Redesign for Improved Usability. In *IEEE Symposiium on Visual Languages and Human-Centric Computing* (VLHCC '08), pages 189–192, 2008.

[88] J. Stylos and B. Myers. Mapping the Space of API Design Decisions. In *IEEE Symposium on Visual Languages and Human-Centric Computing* (VLHCC '07), pages 50–60, 2007.

[89] A. Sutherland and K. Schneider. UI Traces: Supporting the Maintenance of Interactive Software. In *Proceedings of International Conference on Software Maintenance* (ICSM '09), pages 563–566, 2009.

[90] J. Svajlenko, C. Roy, and J. Cordy. A Mutation Analysis Based Benchmarking Framework for Clone Detectors. In *Proceedings of Short/Tool Papers Track of the ICSE 7th International Workshop on Software Clones* (IWSC '13), pages 8–9, 2013.

[91] T. Systa. Understanding the Behaviour of Java Programs. In *Proceedings of the 7th Working Conference on Reverse Engineering* (WCRE '00), pages 214–223, 2000.

[92] T. Systa , K. Koskimies, and H. Muller. Shimba An Environment for Reverse Engineering Java Software Systems. *Software Practice and Experience*, volume 31, issue 4, pages 371–394, 2001.

[93] S. Voigt, J. Bohnet, and J. Dollner. Object Aware Execution Trace Exploration. In *Proceedings of the International Conference on Software Maintenance* (ICSM '09), pages 201–210, 2009.

[94] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Swanson, and J. Isaak. Visualizing Dynamic Software System Information through High-level Models. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA '98), pages 271–283, 1998.

[95] Y. Wu, L. W. Mar, and H. C. Jiau. CoDocent: Support API Usage with Code Example and API Documentation. In *Proceedings of the 5th International Conference on Software Engineering Advances* (ICSEA '10), pages 135–140, 2010.

[96] T. Xie and J. Pei. MAPO: Mining API Usages from Open Source Repositories. In *Proceedings of the International Workshop on Mining Software Repositories* (MSR '06), pages 54–57, 2006.

[97] Z. Xing and E. Stroulia. API-Evolution Support with DiffCatchUp. In *IEEE Transactions on Software Engineering*, volume 33, issue 12, pages 818–836, 2007.

[98] A. Zaidman and S. Demeyer. Managing Trace Data Volume Through a Heuristical Clustering Process Based on Event Execution Frequency. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering* (CSMR '04), pages 329–338, 2004.

[99] A. Zaidman, T. Calderfs, S. Demeyers, and J. Paradaens. Applying Webmining Techniques to Execution Traces to Support the Program Comprehension Process. In *Proceedings of the 9th European Conference on Software Maintenance and Re-engineering* (CSMR '05), pages 134–142, 2005.

[100] I. Zayour. *Reverse Engineering: A Cognitive Approach, a Case Study and a Tool*. PhD dissertation, http://www.site.uottawa.ca/∼tcl/gradtheses/, University of Ottawa, pages 167, 2002.

[101] M. F. Zibran and C. K. Roy. Conflict-aware Optimal Scheduling of Code Clone Refactoring. *IET Software*, volume 7, issue 3, pages 167–186, 2013.

[102] M. F. Zibran, R. K. Saha, C. K. Roy, and K. A. Schneider. Evaluating the Conventional Wisdom in Clone Removal: A Genealogy-based Empirical Study. In *Proceedings of the Software Engineering track of the 28th ACM Symposium On Applied Computing* (ACM SAC '13), pages 1223–1230, 2013.

[103] M. F. Zibran and C. K Roy. A Constraint Programming Approach to Conflict-aware Optimal Scheduling of Prioritized Code Clone Refactoring. In *Proceedings of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation* (SCAM '11), pages 105–114, 2011.

[104] M. F. Zibran, R. K. Saha, M. Asaduzzaman, and C.K. Roy, 2011. Analyzing and Forecasting Near-miss Clones in Evolving Software: An Empirical Study. In *Proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems* (ICECCS '11), pages 295–304, 2011.

[105] M. F. Zibran and C. K. Roy. IDE-based Real-time Focused Search for Near-miss Clones. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing* (SAC '12), pages 1235–1242, 2012.

[106] Software Engineering. *http://se.inf.ethz.ch/old/teaching/2008-S/se-0204/slides/21-Legacy-Software.pdf*. Accessed in 2013.

[107] Tutorial: DTrace by Example. *http://www.oracle.com/technetwork/server-storage/solaris10/dtrace-tutorial-142317.html*. Accessed in 2013.