ASPECT ORIENTED APPROACH FOR TESTING SOFTWARE APPLICATIONS AND AUTOMATIC ASPECT CREATION

Ph.D. Thesis

MANISH JAIN

ID No. 2012RCP9513

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MALAVIYA NATIONAL INSTITUTE OF TECHNOLOGY JAIPUR

August 2018

Aspect Oriented Approach for Testing Software Applications and Automatic Aspect Creation

Submitted in

fulfillment of the requirements for the degree of

Doctor of Philosophy

by

Manish Jain

ID: 2012RCP9513

Under the Supervision of

Dr. Dinesh Gopalani

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MALAVIYA NATIONAL INSTITUTE OF TECHNOLOGY JAIPUR

August 2018

CERTIFICATE

This is to certify that the thesis entitled **"Aspect Oriented Approach for Testing Software Applications and Automatic Aspect Creation"** being submitted by **Manish Jain (2012RCP9513)** is a bonafide research work carried out under my supervision and guidance in fulfilment of the requirement for the award of the degree of **Doctor of Philosophy** in the Department of Computer Science and Engineering, Malaviya National Institute of Technology, Jaipur, India. The matter embodied in this thesis is original and has not been submitted to any other University or Institute for the award of any other degree.

Place: Jaipur
Date:

Dr. Dinesh Gopalani
Associate Professor
Department of Computer Science and Engineering
MNIT Jaipur

DECLARATION

I, **Manish Jain**, declare that this thesis titled, **"Aspect Oriented Approach for Testing Software Applications and Automatic Aspect Creation"** and the work presented in it, are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this university.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this university or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exceptions of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself, jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Date:

Manish Jain

(2012RCP9513)

ACKNOWLEDGEMENT

It is a matter of great pleasure for me to thank all those people who inspired me and gave their kind hearted support at every stage of my research work. First and foremost, I bow in great reverence to almighty God, the most gracious, the most merciful, whose bounteous blessings enabled me to accomplish this thesis.

I would like to express my deep and sincere gratitude to my esteemed guide, Dr. Dinesh Gopalani, Malaviya National Institute of Technology for his invaluable guidance and spending precious hours for my work. His excellent cooperation and suggestion through stimulating and beneficial discussions provided me with an impetus to work and made the completion of this work possible.

My sincerest thanks to all faculty members of Department of Computer Science and Engineering, MNIT Jaipur, for their constant support and imparting me the best of knowledge during my Ph.D. research work.

I would like to thank all non-teaching staff members of Department of Computer Science and Engineering, MNIT Jaipur and all those people whose favours I have received for completing this research work.

Thanks go to my family for providing the support and encouragement that I can always count on. I thank my friends who have directly or indirectly contributed by giving their valuable suggestions.

I am humbled by all the support I received in making this research work a reality.

ABSTRACT

Due to the availability of abundant platforms and the accelerating pace at which the software applications are being developed, testing software for quality assurance becomes extremely important. Software testing is carried out in order to ensure that the developed software conforms to the stated requirements and operates elegantly when put to use in the real environment with divergent operating systems, variety of devices, different browsers and concurrent users. There are various types of software testing classified based on the knowledge of the system, phase at which testing is being performed, extent of automation, source code execution or non-execution, functional or non-functional behavior of the software etc. However, carrying out the different types of testing of software applications manually is quite labour-intensive and error-prone and thus it becomes necessary to deploy automated software testing techniques. This thesis proposes the use of Aspect Oriented Programming (AOP) for automating different types of software testing. Our basis of proposing AOP for the purpose of software testing is that *aspects* in AOP can be used to capture execution points within the program's modules where we suspect bugs and further testing code can be written within the aspect to test such captured components. We establish that AOP alone suffices for carrying out most of the types of software testing and thus obliterates the need of using distinctive tools for different types of testing. One of the major drawbacks of the existing automated testing tools is that the testers need to acquire the required level of proficiency for writing testing codes using these automation tools. In order to reduce the learning curve associated with acquiring the AOP skills, we have also developed a domain specific language, named *Testing Aspect Generator Language (TAGL)* using which the testers, without having the knowledge of AOP, can write the testing codes in the form of simple English-like statements. The TAGL statements are converted into testing aspects using the lexical analyser and parser which we have developed using *lex* and *yacc*. Further we used *AspectJ*, the de-facto standard of AOP, for the testing of widely used open source Java projects like JDownloader, JFreeChart, NetC, JScreenRecorder etc. and found remarkable bugs into them. We posted the discovered bugs to their respective developer teams upon which they assured to debug the reported bugs and make appropriate changes in the future releases of their applications. We have also provided detailed quantitative as well as qualitative comparisons of our approach with the conventional testing methodologies in this thesis. The key benefits of our proposed approach are reduced number of lines of testing code, decreased test execution time and improved source code coverage.

Contents

List of Figures

List of Tables

List of Source Codes

List of Abbreviations

AOP	Aspect Oriented Programming
OOP	Object Oriented Programming
POP	Procedure Oriented Programming
NFR	Non-Functional Requirement
TAGL	Testing Aspect Generator Language
LALR	Look Ahead LR Parser
PARC	Palo Alto Research Center
AJDT	AspectJ Development Tools
ACDT	AspectC/C++ Development Tools
SRS	Software Requirement Specification
GUI	Graphical User Interface
API	Application Programming Interface
JSP	Java Server Page
HTML	Hypertext Markup Language
URL	Uniform Resource Locator
SQL	Structured Query Language
HTTP	Hyper Text Transfer Protocol
LAN	Local Area Network
DSL	Domain Specific Language
GPL	General Purpose Languages
CSS	Cascading Style Sheets
EC	Equivalence Classes
BRD	Business Requirements Document
URD	User Requirements Document
DIDUCE	Dynamic Invariant Detection $\cup$ Checking Engine
GC	Garbage Collection
XML	eXtensible Markup Language
CSV	Comma Separated Values

Chapter 1

Introduction

A program or software is written by humans and humans are bound to make mistakes. As human lives have become increasingly dependent on software, testing the software for quality assurance is paramount. Software testing increases the customer's reliability and satisfaction regarding the quality of product and also ensures lower maintenance cost. But the relative cost of testing a software as compared with the overall cost of developing it, is quite high. Therefore it is important to explore techniques that reduce the testing efforts as well as the inherent complexities of the testing process. We have proposed Aspect-Oriented Programming (AOP) as a mechanism for automating the testing process in an effective way and further established that AOP is solely suitable for performing various types of software testing.

1.1 Software Testing

Software testing is of utmost importance in the software development life cycle for several reasons. Most important reason being that software have become an inevitable part of human life. Statistically looking at all the known utilisation of software in the human life, there have been remarkable aid in the way we can communicate, transact business, and carry out scientific and engineering work. Besides, it is paramount to ensure that a software does not lead to failures because such failures can prove to be very expensive in future and become a cause of rework in the later stages of software development. Furthermore, producing quality software has been identified as the key factor in the success of the organisations and quality can only be assured by testing software for conformance to the specified

design requirements. Any deviations from the specified requirements that emerge up in software are called bugs. Software testing is carried out with the intent of finding out such bugs as early as possible and to fix them prior to the release of the software. The software testing life cycle comprises of identifying, isolating and lastly rectifying the bugs [1].

Software testing is categorised into different types [1, 2]. On the basis of the phase at which it is being performed, software testing is classified as Unit Testing, Integration Testing, Functional Testing, System Testing and Acceptance Testing. In unit testing, a small portion of the application is tested in isolation and its behaviour is verified independently from other parts. Integration testing further ensures that the different parts of the system work seamless when grouped together. Integration testing is done to test the data and control flow among modules and find out the interaction bugs. Integration testing can be done using the Big Bang approach in which all the modules are integrated when they are fully ready and then tested. Another approach for integration testing is the incremental approach in which individual modules are grouped one by one. Incremental approach can be further of two types: Top Down and Bottom Up. Functional Testing is conducted in order to ensure that the software has all the required functionality as specified within its requirement specification. System testing comprises of testing for both functional as well as non-functional requirements. Acceptance testing is the testing performed directly by the end users of the application using the real time production data.

When classified in accordance with the knowledge of the system, software testing can be Black Box Testing, White Box Testing or Grey Box Testing. Black box testing is based solely on the requirements and specifications and is performed without the knowledge of the internal structure of the application. In white box testing, the test cases are prepared based on the knowledge of the internal paths, code structures, and the implementation of the software being tested. Grey box testing is a blend of black and white box testing in which the application is tested by knowing limited information about its internals. Grey box testers depend on the interface definition, internal data structures and algorithms of the application and not on the actual source code.

Corresponding to the extent of automation used, software testing can be Manual or Automated Testing. In manual testing, test cases are executed by the human resources without using any tool or script and therefore it is time consuming. Manual testing is useful only when the test cases are to be executed once or twice,

and frequent repetition is not necessary. Automated testing, on the other hand, is conducted with the automation tool support and becomes advantageous when the tests are to be executed repetitively. For example, for regression testing where code changes frequently, automated testing is the preferred approach as the test cases can be reused.

Furthermore, other types of software testing also exist which are defined based on the procedure used, source code execution or non-execution, functional or non-functional behavior of the software etc.

1.2 Aspect-oriented software development

Aspect-Oriented Programming (AOP) is a new programming methodology beyond the existing software development approaches which is receiving considerable attention from the research and practitioner communities. With the use of AOP, programmers can map the software requirements to the programming constructs in a more logical and natural way and thus aspect-oriented software development has made a profound impact in the area of software development in the past few years. AOP provides a mechanism for separation of crosscutting concerns from the core concerns. A concern is actually a functionality necessary in a software system. Any software system is thus a realisation of one or more concerns. For example, in a typical banking system, the concerns could be Saving Account management, ATM management, Current Account management, Internet Banking, Fixed Deposit management, Customer Care and many more. There are two types of concerns:

- Primary Concern: These are the business logic concerns, also called the core concerns

- Secondary Concerns: These are the system level concerns, also called the crosscutting concerns

Crosscutting represents a situation when a particular requirement of the software is met by placing code into objects (code structures) throughout the system but this code doesn't directly relate to the functionality defined for those objects. The crosscutting concerns neither fit cleanly into the Object-Oriented Programming

(OOP) [3] nor the Procedure-Oriented Programming (POP). In AOP, we introduce a new unit of modularisation - an ***aspect*** - within which we implement the crosscutting concerns instead of fusing them into the core modules [4]. Aspects thus provide a non-invasive way of dealing with the crosscutting concerns. An *aspect weaver* which is a compiler like entity combines the core and crosscutting modules as shown in Figure 1.1 through a process called *weaving*.

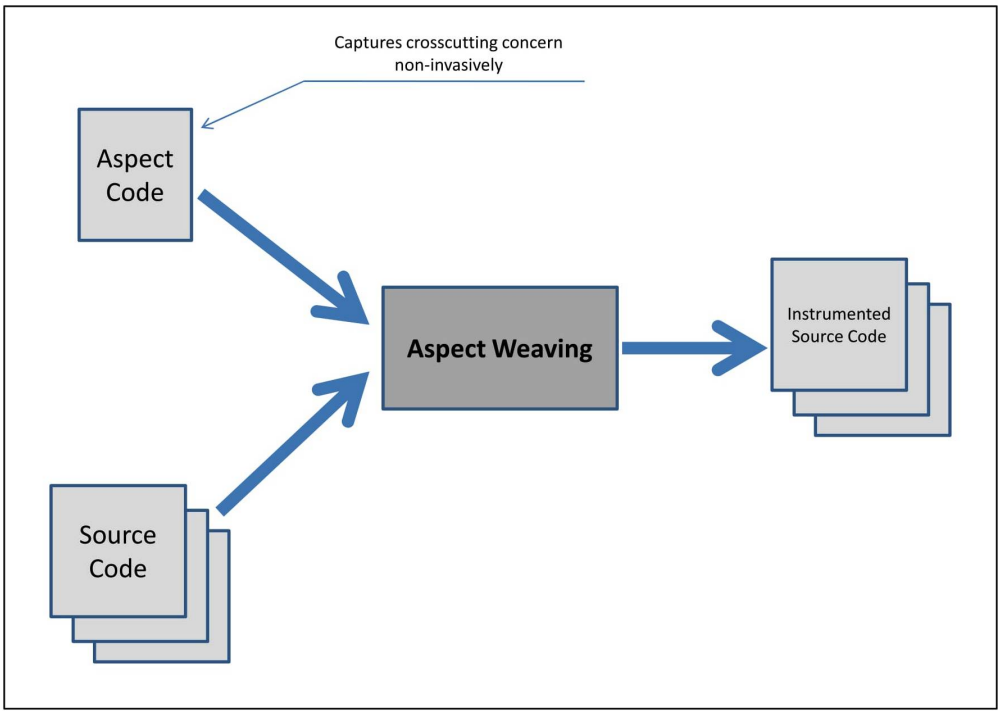

Figure 1.1: Aspect Weaving

1.3 Motivation

Even the best software programmers might leave mistakes in the programs they develop, which can only be caught in the testing phase. Testing adds real value to software and results in high quality product but due to the increasing size of the software, it actually happens to be the most expensive and time consuming step in the software development life cycle. Software testing is an important field of research within computer science and significant work is going on in the direction of understanding the costs of testing and developing effective techniques that detect different types of bugs with least amount of time and efforts.

Following are the key challenges from the field of software testing in which research is ongoing:

- To maximise the test coverage which is a promising measure of test effectiveness that tells us how much of a codebase is covered by the test cases

- Developing new testing techniques that reduce testing efforts and aid for automated test execution, regression testing, test documentation and analysis of test results

- Testing concurrent systems where the system's behaviour depends on the interleaving of the concurrently running events

- To identify the non-functional requirements (NFR) and develop software testing approaches dealing with testable NFRs

- Developing tools that simulate the behaviour of surrounding environment while performing integration testing of a complex system

- Managing the learning curve of software testers so as to facilitate them to become familiar and efficient in the use of the tool

- Improve testability of a system by modeling the development in such a way that testing requires less effort and becomes more effective

The main motive behind our research work is to use AOP and establish it as a technique for carrying out automated testing of software applications which not only reduces the efforts required for testing process but also overcomes various limitations of the conventional testing technologies. Our proposed testing technique addresses almost all of the areas of interest of researchers enlisted above. We have proposed that execution points in the source code can be captured using constructs available in AOP and further the desired testing code can be written within the aspects.

1.4 Aims and Objectives

Few well built testing tools are available for performing testing of functional requirements but for non-functional requirements like reliability, recovery, memory

limitations, security, invariant conditions, logging etc., either their testing has to be done manually or else distinctive tools for each type of testing are required. Moreover, testing a software for fulfilment of such crosscutting requirements requires the original source code to be instrumented at several places. AOP provide us with *aspects* which can be used to capture the desired execution points within the program. These captured execution points can be then instrumented with the testing code written within the aspects. This alleviates the need to alter the source code for the purpose of testing.

A test adequacy criterion is a predicate that defines which attributes of a program should be exercised in order to formulate a comprehensive test [5]. Using conventional techniques, selecting the code that is covered by a test adequacy criterion is not easy because the parts of covered code could be scattered across the program. Use of AOP constructs make it possible to capture multiple execution points of a program simultaneously and thus AOP simplifies the selection of distinct program elements specified in the test adequacy criteria in a non-invasive manner.

Moreover, the conventional testing techniques require the tester to learn and acquire the necessary skills for using the technique. Excessive length of testing code is another issue. In a nutshell, the conventional testing techniques suffer from various shortcomings. Thus, in order to address these issues associated with the conventional techniques, we present AOP testing technique and Testing Aspect Generator Language (TAGL) in this thesis with the following objectives:

1. To identify the applicability of Aspect Oriented Programming for performing various types of testing of software applications.

2. To establish that AOP alone suffices for carrying out most of the types of software testing.

3. To apply AspectJ (the Java-based de-facto standard for Aspect Oriented Programming) for testing of widely used open source projects in Java and attempt detecting bugs into them.

4. To develop a Testing Aspect Generator Language (TAGL) which can be used by the Java application's testers without the knowledge of AspectJ to delineate the testing aspects.

5. To develop a lexical analyser and parser using *lex* and *yacc* that take statements written in our TAGL as input and automatically produce the corre-

sponding AspectJ testing aspects to test the source code of the Java applications.

6. To compare our proposed approach for software testing with the available conventional testing techniques and establish the quantitative as well as qualitative benefits of using the former over the later.

Thus, the main aim of our research is to address the various shortcomings of the conventional testing techniques using aspect-oriented testing approach.

1.5 Contributions

In this thesis, we have presented a novel approach for software testing using aspects in AOP. Using aspects in AOP, execution points in the program under test where bugs are suspected to exist can be captured and the testing code to determine these bugs can be written. We propose that AOP can be used for test automation across the software testing life cycle as depicted in Figure 1.2. Our methodology can be used to automate the generation of test cases, write the test script (using our Testing Aspect Generator Language as explained later in Chapter 5), execute the test cases and further compare the results with the expected results and prepare a test report. Test cases can be provided directly by the tester, or in some cases generated using aspects. For example, aspects can be written to generate random noise to test for concurrency related errors or to generate load by creating multiple users for load testing. Likewise orthogonal, all-pairs or boundary value test inputs can also be generated. Further the code for testing the application under test can be written within the aspects. We have also devised our own domain specific language, named *Testing Aspect Generator Language (TAGL)* using which the testers without the knowledge of AOP can write the test code in the form of English-like TAGL statements which are automatically converted into testing aspects. The testing aspects are weaved with the source code of application under test and then executed. We also used aspects for comparing the actual results with the expected results and to produce a test report specifying the successful and failed tests.

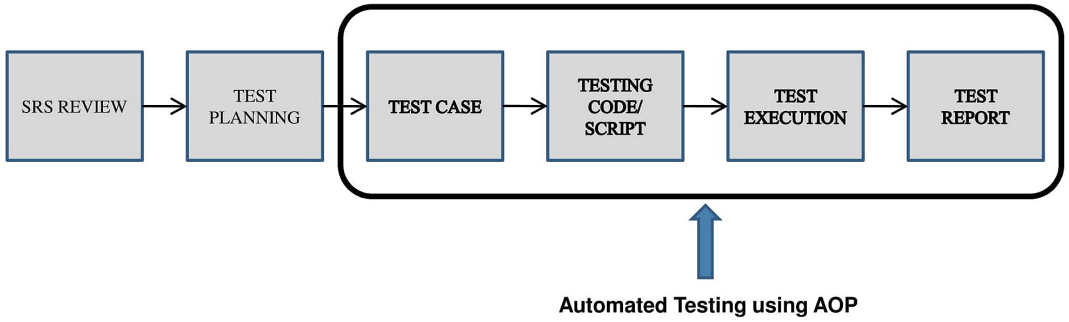

Figure 1.2: Automation across the software testing life cycle

Figure 1.3 shows the sequence diagram of our proposed methodology. The tester writes the testing code in the form of TAGL statements. The expected results can also be specified using our TAGL. Our *Look-Ahead LR parser* (LALR) parser, which has been written using lex and yacc, converts these TAGL statements into the testing aspects. The aspect weaver weaves these testing aspects with the source code under test. Further this instrumented source code is executed and the actual results obtained are compared with the expected results as specified by the tester. Based on this comparison, a simple test report giving details about the successful and failed tests is prepared. Context collection constructs are also available that can be used within the testing aspects to capture the context for failed test cases and such context information later proves to be helpful for debugging.

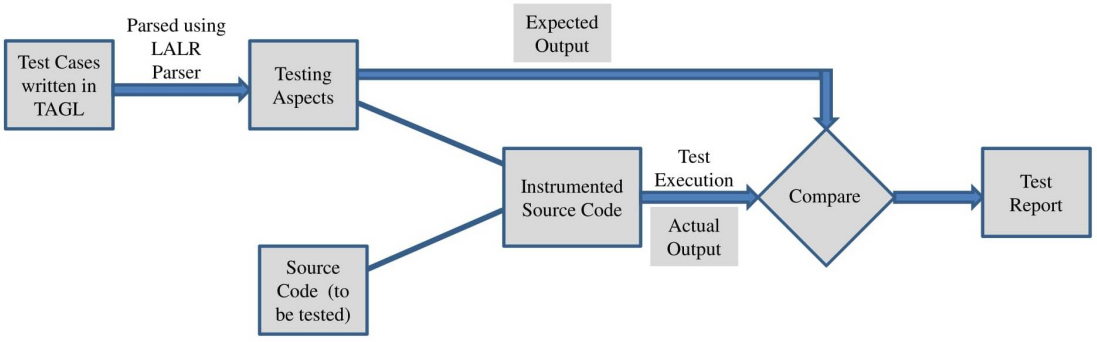

Figure 1.3: Sequence diagram depicting the automatic conversion of TAGL statements into testing aspects which are then weaved with the source code

A summary of the main contributions of this thesis is provided hereunder:

1. We reviewed literature in order to find out the challenges in the field of software testing and areas demanding research. We found that research is going on to develop automated testing techniques which reduce the testing

efforts and provide for provisions like creating surrounding environment for integration testing, regression testing, analysis of results etc. [6, 7]. We have proposed the AOP technique for testing which addresses several shortcomings of the existing testing methodologies and simplifies the testing process.

2. There are numerous tools arising to support testing process which can be used in different areas of testing but it is difficult to distinguish which testing tools should be used in accordance with the organisation's process or the software project. We have proposed the AOP technique which alone suffices for carrying out most of the types of software testing. Using conventional methods, not all types of testing can be carried out using single tool and different types of testing requires distinctive tools [1, 8, 9, 10]. For example, for unit testing Java applications, JUnit would suffice but it does not support load testing or security testing.

3. For carrying out certain types of testing like non-functional testing, the source code is required to be modified [11, 12]. Using our technique, the crosscutting non-functional testing concerns can be captured within aspects and tested and the original source code remains unmodified.

4. Testing for concurrency related bugs like deadlocks or race conditions with conventional techniques requires creating complex testing environment with multiple processors or users [13, 14]. We simplified such concurrency testing using our AOP approach by inducing heuristically controlled sleep using aspects for producing interleavings that might cause errors.

5. Use of AOP makes the selection of code based on a specified test adequacy criteria easy [15]. We used *wild card pointcuts* (explained in Chapter 2) to select scattered code which is to be exercised in order to formulate a comprehensive test.

6. Code coverage is an important metric for analysis of test quality and many development groups advocate high coverage to achieve quality targets [16]. We compared the code coverage of test cases written using AOP technique with that of exiting testing techniques using the *EclEmma* Java code coverage tool and observed that with AOP, we could achieve better code coverage with lesser number of lines of testing code.

7. We applied AspectJ (the AOP extension for Java) on the testing of widely used open source projects like JDownloader, JFreeChart, JScreenRecorder, NetC, JGAP etc. and identified remarkable bugs into them. The bugs

discovered were posted into the bug reporting forums of the respective applications or mailed to their developers and got acknowledged.

8. Domain specific languages are easier for domain experts to use than the general purpose languages [17]. We have developed our own domain specific language, named Testing Aspect Generator Language (TAGL) which can be used by the testers to describe their testing code. TAGL minimises the learning curve and also reduces the number of lines to be written for the testing code. We have written a LALR parser using lex and yacc which automatically converts the TAGL statements into the testing aspects.

9. We used the context collection mechanism of AOP languages [18] to collect necessary context information regarding the execution point where a bug has been identified. This context information proves to be useful while carrying out the challenging task of bug resolution [19].

Publications

The list of findings and publications as a part of this research work are enlisted here:

- M. Jain and D. Gopalani, "Use of aspects for testing software applications," IEEE International Advance Computing Conference (IACC), Bangalore, India, 2015, pp. 282-285. doi: 10.1109/IADCC.2015.7154714

- M. Jain and D. Gopalani, "Memory leakage testing using aspects," International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT), Davangere, India, 2015, pp. 436-440. doi: 10.1109/ICATCCT.2015.7456923

- M. Jain and D. Gopalani, "Aspect Oriented Programming and Types of Software Testing," Second International Conference on Computational Intelligence and Communication Technology (CICT), Ghaziabad, India, 2016, pp. 64-69. doi: 10.1109/CICT.2016.22

- M. Jain and D. Gopalani, "Testing Application Security with Aspects," International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT), Chennai, India, 2016, pp. 3161-3165. doi: 10.1109/ICEEOT.2016.7755285

- Accepted: M. Jain and D. Gopalani, "Domain Specific Language for Automatically Generating Testing Aspects," IEEE International Conference on Emerging Trends in Computing and Communication Technologies (ICETCCT), Dehradun, India, 2017.

- M. Jain and D. Gopalani, "Automated Java Testing: JUnit versus AspectJ," International Journal of Computer and Systems Engineering: International Science Index, Volume 11:11, 2017, pp. 1153-1158. dai:10.1999/1307-6892/100082245

1.6 Thesis Structure

The rest of this thesis is organised as follows. In Chapter 2, we have described the fundamentals of our proposed AOP approach, listed out languages whose AOP extensions are available and also explained the importance of software testing. Further in this chapter, we provide a brief description of the related work carried out in the domains which are connected to our proposed technique. Thereafter in Chapter 3, we describe the suitability of applying AOP for performing different types of software testing with appropriate illustrations. In Chapter 4, we have described the application of AspectJ as an example AOP language, to perform different types of testing of certain popular open source Java software and detected phenomenal bugs into them. Details of the deployed open source software, their usage and features are given. In Chapter 5, we have explained the concept of domain specific languages and proposed our own domain specific language, named Testing Aspect Generator Language which can be used for automatic creation of testing aspects. This chapter provides an explanation of the syntax of the TAGL which is simple to learn and use and can be used by the testers for describing their testing code. We have also provided details of our lexical analyser and parser which have been developed using lex and yacc respectively and convert the TAGL statements into testing aspects. Chapter 6 and 7 evaluate our AOP approach for testing by providing a detailed comparison with the conventional testing techniques. The qualitative and quantitative benefits of using our AOP approach for testing have been listed. Chapter 8 provides the conclusions drawn based on the work with directions for possible future work. At last, we have provided a list of the references.

Chapter 2

Background and Related Work

Aspect Oriented Programming (AOP), like Object Oriented Programming (OOP) and Procedure Oriented Programming (POP), is a programming methodology in its own right. AOP aims at separating the secondary concerns like logging, security, authentication etc., which are crosscutting in nature, from the primary business logic. In this chapter, we will first discuss the fundamentals of aspect oriented programming and then the importance of software testing in detail. Important AOP languages are described with suitable illustrations. Then after, this chapter reviews the work in the field of automated software testing techniques and their shortcomings and further the use of AOP for software testing.

2.1 Fundamentals of Aspect Oriented Programming

Cristina Lopes and Gregor Kiczales who hailed from Palo Alto Research Center (PARC) which is a subsidiary of the Xerox Corporation participated in the initial development of Aspect Oriented Programming (AOP). Gregor was the leader of the team that developed the original AspectJ at PARC and also gave the term "AOP" in 1996 [20]. Currently, *eclipse.org* which is an open source community is providing updates and maintaining support for the AspectJ project.

AOP is a programming methodology which improves the modularity of the software by separating the crosscutting concerns from the core concerns. A crosscutting concern is either a characteristic expected from the software like security or it could be a behaviour which one or all classes must exhibit like logging. Crosscutting concerns are not correlated to the main business logic of the application. If

we take example of a banking application, handling the various type of accounts, managing the transactions, interest calculation, online banking services shall be the primary concerns which will form the main business logic. But at the same time, there would be certain concerns like **logging** to avoid accidental data loss, **security** to check authorisation of users, **caching** for performance optimisation that shall span across multiple modules. These system wide concerns that span across multiple modules are called crosscutting concerns.

The scenario of crosscutting is inherent in most complex systems. It can lead to issues like code tangling or code scattering. Code tangling refers to the phenomenon when a single module or component in an application contains code related to multiple concerns (see Figure 2.1). For example, a business logic module in a banking application implemented using OOP methodology might contain code for security, logging and recovery as well.

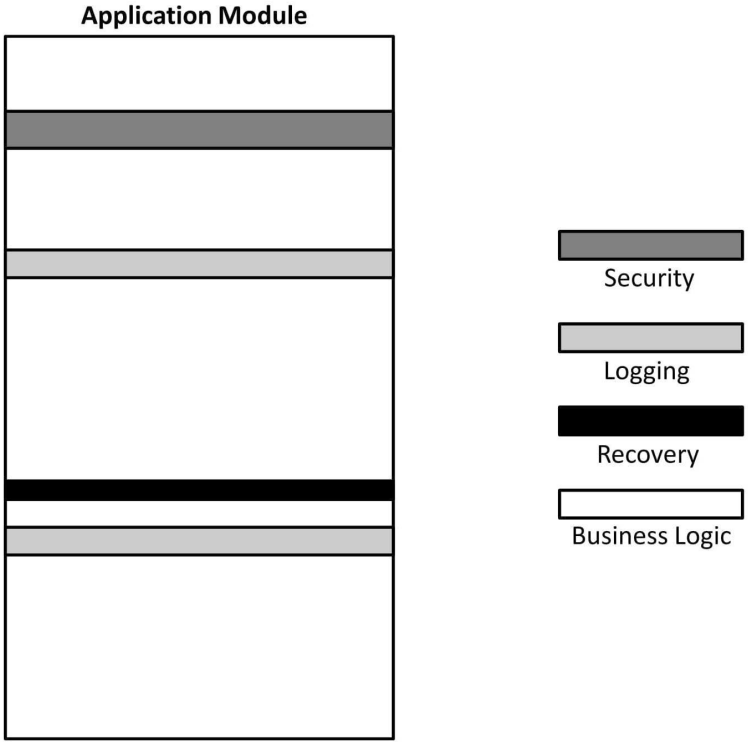

Figure 2.1: Code Tangling

Code scattering is the term given to the situation when the code for a concern is spread across multiple modules or components (see Figure 2.2). For example, the code for logging concern, which logs the execution of each account transaction in a banking application, shall be spread across various modules of the application like in the internet banking module, deposit module, withdrawal module, ATM

module etc.

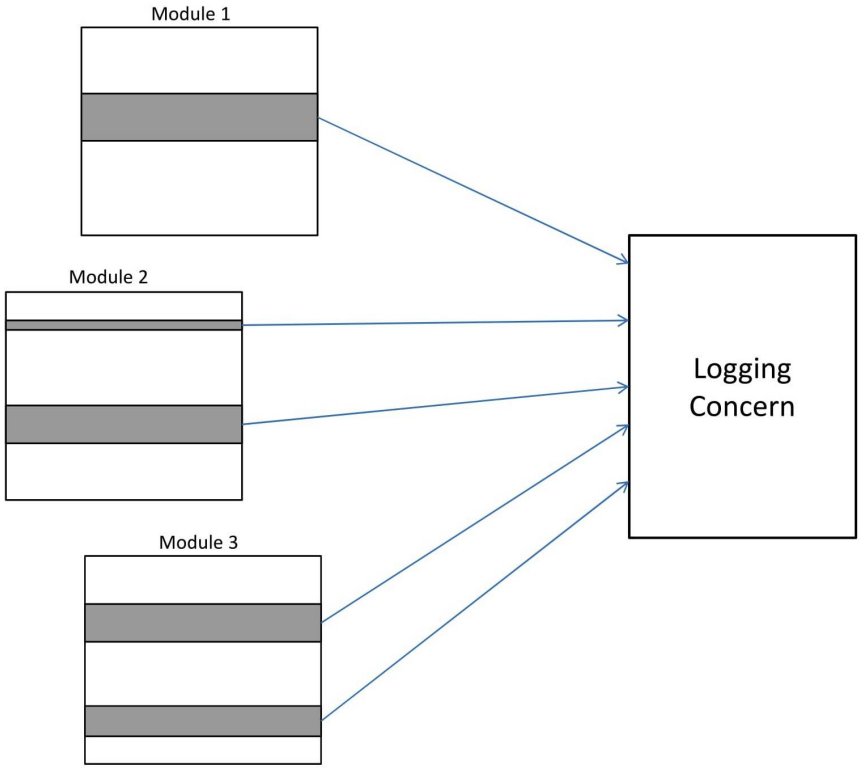

Figure 2.2: Code Scattering

Although OOP is a good methodology for managing the core concerns but it does not provide any mechanism to handle the crosscutting concerns in a modular way. With OOP, these system wide crosscutting concerns are spread throughout the application and cannot be modularised. The basic idea of AOP arises from the fact that the concerns that are crosscutting too have a clear purpose and therefore they can be modelled into certain programming structure in a modular way. *Aspects* in AOP are a new unit of modularisation that provide a mechanism to modularise the crosscutting concerns. The crosscutting concerns are implemented inside the aspects in order to avoid them from intermixing in the business logic modules.

AOP is becoming an increasingly popular programming methodology. AOP is being used by developers in the real projects to introduce features like security, logging, for improving the performance, for providing data transfer functionality, for implementing authorisation rules and so and so forth. Use of AOP in the real projects has produced impressive results as it enhances the modularity of the code and at the same time reduces both the amount of code and time required to create the software [18]. AspectJ which is the aspect oriented extension of Java has got wide acceptance because the latest version of AspectJ (1.8.10) has got

all the required functionality that a language should possess in order to make it possible to work with real life projects.

AOP is actually built on top of the existing OOP and POP technologies and thus expands them with the capabilities to address the system wide spread crosscutting concerns. Thus, developers which use AOP implement the primary concerns using the base methodology only and then modularise the crosscutting concerns using AOP constructs. There are a total of four constructs in AOP [18] which form the basis to specify the crosscutting concerns in a modularised way. We discuss them hereunder:

- **Aspect** - The aspect is the central unit, just like the class in OOP languages. It contains the code that expresses the weaving rules.

- **Jointpoint** - A joinpoint is any identifiable point in the execution of a program.

- **Pointcut** - A pointcut is a construct that selects the joinpoints.

- **Advice** - Advice is the code to be executed at the joinpoint captured by the pointcut. Advice can execute *before*, *after* or *around* the joinpoint.

All of the above four are the building blocks that form the basis of our proposal to use AOP for software testing. *Aspects* can be used to capture one or more execution points using *pointcuts* which can further be tested by writing testing code within *advices*.

2.2 Aspect Oriented Languages

Although AspectJ is the most popular aspect oriented language extension meant for Java, AOP implementations for other popular programming languages are also available. In what follows, we discuss the AOP implementations for the widely used programming languages.

2.2.1 AspectJ

AspectJ is a general purpose aspect-oriented extension to Java that offers a great deal of power and improved modularity to the programmers. AspectJ development can be made easier using the Eclipse AspectJ Development Tools (AJDT) plug-in. A wizard is provided for creating a new AspectJ project or we can also convert an Java project into an AspectJ project by right-clicking the project from the package explorer view. AJDT provides us with the Cross Reference view which exhibits the crosscutting relationships for the various program elements. AJDT also supports the annotation style of declaration, known as @AspectJ style.

In AspectJ, we have aspects within which we write the *pointcut* to capture the desired execution points and *advice* to write the code to be executed at the captured points. We have *before*, *after*, *after returning*, *after throwing* and *around* advices in AspectJ. The code snippet in Listing 2.1 shows an example of around advice which executes around the *withdrawal* method of the *AccountClass* to confirm that the balance is more than the withdrawal amount.

Listing 2.1: AspectJ: Example Aspect

```
public aspect withdrawalAspect
{
        pointcut checkBalance(int amount, AccountClass account) :
                call(boolean AccountClass.withdrawal(int)) &&
                    ↪ args(amount) && target(account);

        boolean around(int amount, AccountClass account) :
            ↪ checkBalance(amount, account)
        {
                if (account.balance < amount)
                {
                        return false;
                }
                return proceed(amount, account);
        }
}
```

AspectJ also provides us with wildcard pointcuts which can be used to capture multiple execution points having different signatures, return types, arguments etc. simultaneously. For example, in order to cover calls to all the methods of a particular class and all its subclasses irrespective of their return types and number of arguments, we can use a wildcard pointcut like *call(* ClassName+.*(..))*.

2.2.2 AspectC++

The latest version 2.2 of AspectC++ was launched on 10th March'2017 [21]. It is an aspect oriented extension for C and C++ languages. AspectC++ has a source to source compiler which translates the code written in AspectC++ into C++ code. Regarding syntax and semantics, the constructs in AspectC++, namely, *aspect, joinpoint, pointcut, advice etc.* are quite similar to that of AspectJ. Further, there is provision of *introduction* of new data members or methods into one or more classes which are matched by the given pointcut expression. Alike AspectJ, AspectC++ pointcuts also support wildcards. The aspect can be also be ordered using the *order advice*. Aspect inheritance is also supported. AspectC++ plug-in named *AspectC/C++ Development Tools (ACDT)* is available for Eclipse.

Listing 2.2 shows how an aspect in AspectC++ can be used to separate the concern "check whether the stack is full" from the Push method. The logic to check whether stack top has reached is written in the aspect within an around advice.

Listing 2.2: AspectC++: Example Aspect

```
#define MAX 100
aspect stackPush
{
        //st captures the Stack object in context
        pointcut isFull(Stack st) = call("void Stack::Push(int)") &&
            ↪ target(st);
        advice isFull(st): around(Stack st)
        {
                if(st.top == MAX-1)
                {
                        cout<<"Stack is full!";
                        return;
                }
                else
                {
                        tjp->proceed();
                }
        }
};
```

2.2.3 AspectMatlab

The team of Toheed Aslam, Jesse Doherty, Anton Dubrau and Laurie Hendren at Sable Research Group developed AspectMatlab as an aspect oriented language for MATLAB which is a dynamic language commonly used by the scientists [22]. Alike classes, *aspects* in AspectMatlab can have properties and methods. AspectMatlab provides us with *patterns* which are meant for capturing the execution points in the source code and named *actions* within which we write the code to be executed at the captured execution point. Unlike AspectJ, actions in AspectMatlab are declared with a name. Patterns can not only be used to capture *call* and *execution* but also support *get* and *set* patterns to deal with arrays. For example, the pattern *set(arr)* can be used to match all the assignment statements to an array *arr*. Also there are *loop, loophead, loopbody* patterns to capture the executions of both for and while loops which are important structures in scientific programs. Further, there are *selectors* which can be used to collect context information.

The team developed an *amc compiler* which translates the source code written in AspectMatlab into MATLAB code that can be run on any MATLAB platform. Listing 2.3 shows an example of how aspect in AspectMatlab can be used to capture call to a function *MethodDoubleArguments* with two arguments using *patterns*. The aspect further bypasses the execution of the instrumented method using an around advice inside *actions* and executes another function *MethodSingleArgument* which takes only one argument in place of it, with the first of the two arguments captured using *args*.

Listing 2.3: AspectMatlab: Example Aspect

```
aspect myAspect
patterns
        callMethodDoubleArguments : call(MethodDoubleArguments(*,*));
end

actions
        actcall  : around callMethodDoubleArguments : (args)
                   MethodSingleArgument(args{1});
        end
end
```

2.2.4 Aspect Python

For python, we have *aspectlib* aspect oriented programming library which can be used to bring about desirable changes in the behaviour of existing python code. The latest version of the library is aspectlib 1.4.2 which was released on 10th May 2016 [23]. The aspectlib library provides two core tools to do AOP: *aspect* and a *weaver*. An aspect can be created by decorating a *generator* function. The generator yields advices which are used to bring about the desirable behavioural change in the original python code. An aspect instance is a simple function decorator and decorating a function with an aspect will change the function's behaviour according to the advices yielded by the generator. Using the advices available in aspectlib, we can call a function multiple times, or with different arguments or even make a function return a desired value. The weaver patches the classes and functions in the python source code with the given aspect. The aspectlib library also provides with *aspectlib.test.mock* and *aspectlib.test.record* which can be used for the purpose of testing.

The aspect in Listing 2.4 written using the aspectlib library bypasses the execution of the method it instruments. In this example, when the function *fileread* is called with any file name as argument, its bonafide execution is bypassed and the function is executed with the file name (/home/filename.txt) as specified in the instrumenting aspect.

2.2.5 AOP-PHP

AOP-PHP (a PHP Extension Community Library extension) is the easiest way of integrating AOP with PHP. Besides AOP-PHP, there exists many other implementations of AOP in PHP like FLOW3, aspectPHP, Go! etc.

AOP-PHP version 0.2.2b1 is the latest which was released on 18th November' 2012 [24]. AOP-PHP provides us with advices like *aop_add_before*, *aop_add_after*, *aop_add_around* that can intercept the function calls and perform a desired action like modifying arguments during execution or changing return variables when the execution of the function has been completed. We can use the *getArguments()* method to grab the original arguments, modify them and then pass new arguments using the *setArguments()* method. AOP-PHP also allows for wildcard pointcuts to simultaneously match a bunch of methods.

Listing 2.4: Aspect Python: Example Aspect

```
import aspectlib

@aspectlib.Aspect
#here *k means any number of arguments in the advised function.
def bypass_fileread(*k):
    #yield is like return only
    #Proceed bypasses the execution of the instrumented method and executes
        ↪ it with the specified arguments
    yield aspectlib.Proceed("/home/filename.txt")

def fileread(name):
    #this function simply opens the file with name "name" and returns its
        ↪ contents to the caller
    return open(name).read()

#we weave the aspect with the advised function using weave method in aspectlib
from aspectlib import weave
patch = weave(fileread, bypass_fileread)
```

Listing 2.5: AOP-PHP: Example Aspect

```
<?php
class person
{
    public function displayName($name)
    {
        echo "{$name}";
    }
}

function greetingAdvice(AopJoinPoint $joinPoint)
{
    $args = $joinPoint->getArguments();
    $str = 'Hello';
    $args[0] = $str.' '.$args[0];
    $joinPoint->setArguments($args);
}

aop_add_before('person->displayName()', 'greetingAdvice');

$p = new person();
$p->displayName('John');
```

Listing 2.5 shows the basic syntax how an advice can be added to execute before the function of a class. In this example, we ensure that each time the method *displayName* of class *person* is called using its object, AOP-PHP will execute the function *greetingAdvice* before the called method. Inside the function *greetingAdvice*, we use *AopJoinPoint* to capture the advised function *displayName*'s argument and change it such that a "Hello" greeting is added before the name. On the same lines, AOP-PHP can be used to implement complex crosscutting system concerns too. For example, by using an appropriate *aop_add_before* advice, we can ensure that the authorisation permissions of a session user are always checked before a particular function is called.

Here we would like to propound that there exists AOP implementations for other programming languages as well like Ruby, LISP, Perl, Ada, Unified Modeling Language (UML), .NET framework languages etc. but as their AOP implementations are less popular (due to limited AOP constructs), we shall use examples only from the aforementioned AOP languages in the following chapters.

2.3 Importance of Software Testing

With numerous organisations like banking, educational institutions, automobile industries, smartphone companies etc. becoming dependent on software applications for their business, research, development and for providing prompt and satisfactory services to their customers, software that do not work as expected can have a large impact on an organisation. Faulty software can put the organisations into problems of varying nature like loss of time, loss of business or even human hazards in case of safety-critical systems. Therefore it becomes important that the errors are caught well in advance before deploying the software in the real production environment.

The important reasons that make testing of a software essential are explained hereunder:

- To err is human: Software are coded by human beings and human beings commit mistakes in any process. It is always good to identify the fault and errors in the software caused by human mistakes during its development phases. Testing is the best way to identify our mistakes and rectify them before they end up costing us. As a classical example of human error leading to defects, there could be a situation when a developer while developing a

new feature into an application may simply forget and break a legacy feature. Regression testing can catch such errors before the application is put to use.

- In order to ensure that the software meets the expectations as specified in the requirement specification: Regardless of the development methodology, the ultimate goal of testing is to ensure that a software works as per the user expectations. As an initial step of the Software Development Life Cycle, a Software Requirement Specification (SRS) document is prepared which precisely defines the expectations and understanding of a customer's software requirements. The SRS depicts various types of requirements of the system like functional, performance, resources, maintainability etc. and therefore the Test Plan is prepared on the basis of SRS [25]. The test plan enumerates the various test cases that are required to ensure that the features of the SRS have been implemented bug free. Further, it is also important that testers understand every detail specified in the SRS in order to avoid faults in the test cases and their expected results.

- In order to ensure that a software works fine in the real environment with different operating systems, devices, browsers: In the current era of technology, we have variety of devices ranging from tablets, smartphones, laptops, desktops etc. which run over different operating systems. Further in order to access web based applications, there are several web browsers available which become a big challenge in their design and development. Therefore, testing ascertains that the application is compatible on multiple platforms and ensures that its functionality remain unaffected on variety of operating systems, devices and browsers.

- In order to evaluate the application's behaviour under load of several users: An application may work fine with one user but it may not render expected results when hundreds (or more) of users use it simultaneously. A software should always accomplish what it is expected to do, no matter how many users use it. Load testing is conducted by putting simulated demand on an application in order to determine its behaviour under load conditions. There are various automated tools like *Apache JMeter*, *Load Runner* etc. which help in performing load testing of applications. Upper limit of database, mismanagement of memory, buffer overflow, delayed response etc. are common issues which arise when multiple users hit an application simultaneously.

- Producing quality software has been identified as the key factor in the success of the organisations: Delivering quality software after proper testing

increases end user's satisfaction and helps in gaining their confidence. Bugs like the one which compromises user's privacy or crashes the computer or leads to denial of access or service etc. have a measurable adverse impact on the customer satisfaction. A product free from anomalies, which can only be ensured by its thorough testing, invites lesser number of complaints and thus greater customer satisfaction [26]. Organisations that produces well tested quality solutions are likely to get more recognition and more customers.

It is odd but true that the time and cost incurred for software testing are often comparable to that incurred for the software development. Software testing costs usually lies between 25 to 40 percent of the total project cost. Further, Beizer [27] reported that "*half the labour expended to develop a working program is typically spent on testing activities*". But at the same time, delivering well tested quality software to the client is inevitable. Therefore it is important to explore techniques that minimise the efforts required for testing and improve the overall testing process. In this direction, we propose the use of Aspect Oriented Programming methodology for performing automated software testing.

2.4 Literature Review

We identified and read the key papers of research work related to our topic. The purpose of the review was to understand the limitations of the conventional software testing methodologies and further conceive AOP as a solution to address the same. Literature review carried out in three different areas related to our approach and the findings thereof are discussed in detail hereunder.

2.4.1 Conventional Automated Testing

Hooda et. al [1] in their research paper on testing types and techniques have specified the development of a generic testing framework as a work of future scope by stating "*a research and study can be done on the software testing to propose a generic testing framework and techniques to support functional, performance and security testing for object oriented development framework*". Bamotra et. al [8] have stated that "*Various types of tools are used for automated testing and they can be used in different areas of testing*". Mustafa et.al [9] noted that there are many testing tools which can be used for different types of testing. For many years,

researchers and practitioners have proposed a variety of testing tools which can be used to automate the testing process. They have classified these testing tools based on their intended usage. They collected 135 software testing tools from the internet, studied and classified them into different types. During their research, they also observed that testing tools for certain testing methods, for example security testing of application software products, are quite limited and restricted. Similarly, the main objective of Uspenskiy in his Master's thesis [10] was to describe the software testing tools and their corresponding use. He has provided a presentation of validation activities and classification of supportive software tools. He came to a conclusion that the tools are often designed to support one or more software testing methods and are varying in scope from supporting individual tasks to covering the complete testing cycle. However, the AOP approach proposed by us is a versatile technique which supports numerous software testing methods like unit testing, integration testing, load testing, invariant testing, security testing etc. and covers almost every important phase across the software testing life cycle as shown in Figure 1.2 of Chapter 1.

Rafi et. al [28] in their review paper investigated the views regarding the benefits and limitations of test automation. They projected tool selection, automation setup and training the testers as the main limitations of automated testing. After conducting a survey of the practitioner's view of software test automation benefits and limitations, they have stated that the tester should have enough technical skills to build successful automation. To overcome such limitation while using our proposed approach of AOP for software testing, we have developed our own domain specific language, named Testing Aspect Generator Language using which the tester without any expertise of AOP can still write the testing code with ease. The authors also stated that most of the testing tools available in the market are incompatible and do not provide what you need or fits in your environment.

Srivastava et. al [29] have mentioned in their paper that the adequacy of a test is determined by the source code coverage measure that describes the degree to which an application's source code has been tested. Source code coverage analysis is the most mentioned quantitative metric for the assessment of testing process. They further state that if a coverage is not met, then more test cases have to be designed to test the items that were missed and increase the coverage. We have shown in our work that AOP provides better source code coverage based on one or more test criteria with lesser number of lines of testing code.

Shivaprasad et. al [30] worked upon the unit testing of concurrent Java programs

and found that conventional unit testing practises focuses on testing program modules sequentially and are likely to miss concurrent bugs. They have mentioned that even the most widely used unit testing frameworks for Java - like JUnit or TestNG - do not provide good support for testing concurrent issues. We simplified concurrency testing using our AOP approach by inducing heuristically controlled sleep using aspects for producing interleavings that might cause errors.

2.4.2 Testing using AOP Techniques

Duclos et al. [12] have pointed out that making changes to the source code of application for conducting testing is an issue because this can modify their behavior. For example, in order to test for the presence of memory leakages in C++ which arise when the programmer forgets to call the destructor for every constructed object of the class, the tester needs to add counters at several places in the program. An increment is made to the counter when the object is constructed and likewise an decrement is made whenever it is destructed. A positive value of the counter at end of the program indicates a leak of memory. The authors used aspects in AspectC++ to capture all the points in the program where an object is created or destructed and further increased or decreased the value of the counter within the advice without altering the source code directly. Using an aspect, they found a memory leak in a complex C++ software program, NOMAD, which is used in both industry and research.

Bruel et. al [31] defined an approach, covering the whole development life cycle, to incorporate testing functionality into aspect-oriented components. The paper has focused on contract testing in which the components of a system can validate that the servers to which they are "plugged" dynamically at deployment time will fulfil their contract. They describe testability as a non-functional requirement which is better suited to be implemented as an aspect. They have proposed each test to be implemented as an aspect. They have provided an illustration wherein they used the aspects to introduce attributes to the *STACK* class like *IsEmpty* method and also to provide particular values for parameters the user wants to test.

Stamey et. al [32] suggested the use of AspectJ for implementing non-invasive debugging for Java Programs without altering the source code. In their paper, they proposed the use of AOP technique for tracing the execution of loops, methods and constructors with suitable illustrations and implemented two important elements of debugging, namely variable tracing and inspection. The use of aspects

to implement debugging eliminates the learning curve to install and use the new debugging packages for code tracing.

Yang [33] in his article has proposed an AOP-based alternate white box test strategy. He replaced the traditional way of adding test hooks and logs for white box testing by aspects. As a case study, he worked upon ASP.NET MVC 4 web application and conducted essential tests in the areas of security, localisation and content using AOP as the test infrastructure. He claims to have created a test automation engine using AOP that can be used throughout the product development life cycle.

Pesonen et. al [34] while performing testing of Symbian OS observed that one problem with conventional testing techniques is that although it is easy to generate test cases automatically, not all the generated test cases are important and often include huge number of unnecessary ones. An insight is required to distinguish between the important and the unnecessary test cases. However, from his use of aspects for testing the Symbain OS, he observed that AOP provides a variety of different expressions with combination of pointcut which makes aspects an impressive tool for capturing the testability concerns.

Metsa et. al [35] discussed those non-functional requirements of applications which can be more efficiently tested using AOP approach than the conventional techniques and recommend aspect-orientation as a technology which has got great potential to facilitate the automated execution of tests. Although there are several established tools for performing functional testing, comparatively less support exist for testing non-functional requirements.

Xie [36] used AOP to conduct the test process automatically according to the given test data. He argued that as aspect can effect the behaviour of a module, it lays a good foundation for software test. He created a driver module as a mock of the calling module using the around advice which contained the test cases and the loop test conditions. Further, the development process can be adjusted well in time according to the mock feedback.

All of the above discussed literature signify preliminary ideas regarding the use of AOP as an alternative testing strategy for one or the other type of software testing and indicate that with regular advancement in aspect oriented software development and methodologies, advance test automation solutions can be conceived using AOP. We have proposed AOP as a complete solution for carrying out various types of software testing and established its usefulness with experimental

results in this thesis.

2.4.3 Additional Related Work

Metsa et al. [37] suggests that testability of a software system can be increased by using Aspects instead of Macros, which are used to enable the inclusion or exclusion of test code or even Interfaces, which provide easy access to the components hiding their internal implementation. They have stated that aspects can be used for insertion of new variables which can record the system states at various instances and also that the removal of test related functions implemented using aspects is trivial. Sioud in his Master's thesis [38] has worked upon implementing the missing garbage collection mechanism in C++ language using its aspect oriented counterpart AspectC++. Wehrmeister [39] proposed an approach combining Model Driven Engineering and aspect oriented programming concepts so that the functional and non-functional requirements can be dealt in a modularised way. He made use of aspects and developed a Distributed Embedded Real Time Aspect Framework wherein each of the non-functional requirements (NFRs) were handled using an aspect. Upgrading this idea, we have proposed the use of aspects for performing testing of non-functional requirements like security, recovery, performance etc. Pesonen [40] used aspects to measure the product line correctness and efficiency of the embedded system. He has recommended the use of aspects for implementing and handling the new features of expanding embedded product families.

Other than testing using AOP methodology, testing aspect oriented software itself has also been a matter of interest for many researchers. Because of its new construct and properties like weaving etc., AOP brings about new challenges which are not present while testing programs from other programming paradigms like OOP. Ghani et.al [41] in their work explored various ways of testing the aspect oriented programs. They have discussed various issues on testing aspect-oriented programs, presented the fault models and fault types for aspect-oriented programs and proposed automated tools for testing them. Sokenou [42] considered two aspect oriented languages, AspectJ and ObjectTeams/Java and proposed an approach for testing aspects using aspects. On the other hand, we have used the AOP domain itself and verified its suitability for testing various software applications written in Java, C++, Python, Matlab etc.

2.5 Summary

In this chapter, fundamentals of Aspect Oriented Programming and various available Aspect Oriented Languages which lay the basis for discussions and illustrations in the forthcoming chapters were described. We discussed about the importance of software testing and stated the five principal reasons that make software testing essential. Further, we provided the literature review wherein based on the related work carried out in the field of software testing, we explained the various limitations and shortcomings of the conventional testing methodologies. Related work in the area of testing using AOP has also been probed into.

Chapter 3

Proposed Aspect Oriented Approach for Software Testing

There are various types of software testing classified based on the knowledge of the system, phase at which testing is being performed, extent of automation, source code execution or non-execution, functional or non-functional behavior of the software etc. as we explained in Chapter 1. In this chapter, we identify the applicability of AOP for performing different types of software testing. AOP languages provide us with pointcuts which can be used to capture joinpoints from the program code. Using suitable pointcuts, we can capture joinpoints of interest from the program code which need to be tested. There are wildcard pointcuts available in AOP languages which can be used to capture multiple joinpoints that are to be tested simultaneously as discussed in Chapter 2. Further, advice can be used to write the appropriate testing code which shall be executed before/after/around the captured joinpoints and attempt to discover bugs.

We propose the application of AOP to carry out different types of testing efficiently and render most of the illustrations using AspectJ as it is the most developed and popular AOP language so far.

3.1 White Box Testing

White-box testing which is also known as clear box testing, glass box testing, transparent box testing and structural testing, is a technique based on the analysis of the internal structure of the system. The mechanism of white box testing is

shown in Figure 3.1. It focuses on the flow of input and output through the application. After understanding the code of the system under test, bugs due to internal security holes, memory leaks, poorly structured coding paths, improper functionality of conditions and loops etc. are discovered in the process of white box testing. The principal advantage of white box testing is that the testing of the system can be started at an early stage as there is no need to wait for complete development or a completed Graphic User Interface (GUI).

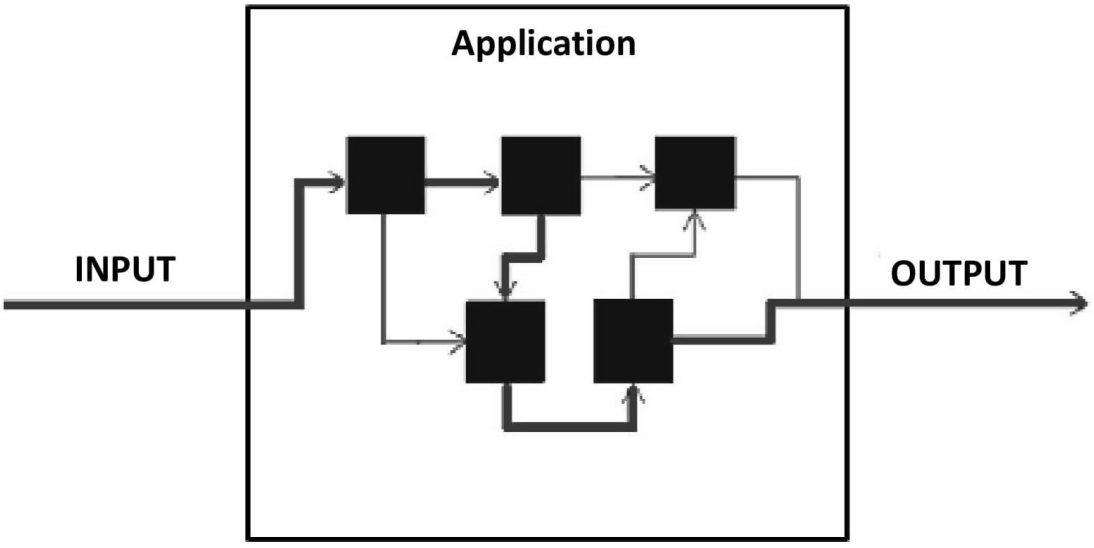

Figure 3.1: White box testing [43]

3.1.1 Aging Testing

Software exhibits a common phenomenon that closely resembles human aging. A software system may slow down with time caused by failure to release the allocated memory or because a routine does not release the complete memory that was allocated. Another reason could be that the strategy of releasing the acquired resources while execution is not fail safe. Such problems are usually caused by the programmer's mistake of forgetting to write the code for clean up or releasing the resources. Aging testing, also known as age testing, evaluates a software's ability to perform in the future. In aging testing, primarily the software is tested to discover issues related to memory leaks because as a result of successive memory leaks arising from application's execution over time, problems like high response times or system crashes surface up [44].

AOP is well suited to find memory leaks. Lets take example of memory leak in an Java application. When a programmer provides for its own finalization method

named *finalize()* to carry out important clean up duties before destructing a class' objects but if he/she forgets to call this self written method and few objects of the class are left *undestructed*, then it leads to a situation of memory leak.

The aspect as shown in Listing 3.1 can be used to find out the existence of such a memory leak. In this example, the pointcut *creation()* captures the call to all the constructors for the class *person* and increments a static counter by one whenever a constructor is called. Similarly the pointcut *destruction()* captures the calls to the finalize method written by the programmer for clean up activities and decrements the counter by one whenever it is called. If the value of the static counter is greater than zero, then it means that the programmer forgot to call the finalize for one or more constructed objects and it reflects a memory leak.

By recording the context information at the time of creation and destruction in suitable data structures, the objects left to be destructed can be exposed. AspectJ provides a special reference variable, *thisJoinPoint* that contains useful context information about the current joinpoint. We used two arrays of string to store the created objects with their source location (using AspectJ context collection construct *thisJointPoint.getsourcelocation().toString()*) and the destructed objects. At the end, the two arrays were compared to spot the undestructed objects' name and their source locations.

Java allows us to define and use objects of a class inside another class and at times, application requirement makes it necessary for the programmer to do so. For e.g. a programmer may define a point class's object within a line class. We can have as many objects of different classes within a class definition. Different memory blocks are allocated for all such objects within the class and all such objects may have different lifetimes [45].

When we use object of a class within another class and it is not required for the complete life cycle of the later class, the memory allocated for the former class is not made free up to the time the object of later class is live. This is a situation of memory leak which cannot be detected by available tools for memory analysis. It is so because this memory leakage is caused due to a logical mistake of the programmer which is hard to discover. Although such memory leak may appear small, but it may gradually increase over time if the application is run continuously and this unused block of memory adds up. The effects of this type of memory leakage shall be more pronouncing if the class of unreferenced object has high memory requirements. In that case, the big memory block allocated for the object shall remain engaged until the scope of the later class is finished.

Listing 3.1: Aging testing using AspectJ-Example I

```
public aspect testMemoryLeak
{
        static int count=0;
        pointcut creation(person obj) : execution(public person.new(..))
            ↪ && this(obj);
        after(person obj) : creation(obj)
        {
                count++;
                //Record object name into suitable data structure for further
                    ↪ analysis
        }
        pointcut destruction(person obj) : execution(protected void
            ↪ person.finalize(..)) && this(obj);
        after(person obj) : destruction(obj)
        {
                count--;
                //Record object name into suitable data structure for further
                    ↪ analysis
        }
}
```

We used aspects to identify such memory leaks which are caused due to objects created within a class but not dereferenced after their use is over. If object of a class say *ClassB* is created within another class say *ClassA*, then whenever object under execution is that of *ClassA*, we increase a counter and when object of *ClassB* is used, we reinitialise the counter to zero. A positive value of the counter indicates that use of object of *ClassB* was finished before the life cycle of *ClassA* and thus it can be dereferenced earlier in the code of the class. Listing 3.2 shall make our point clear.

An array out of bound is another memory related issue which occurs when the index of the array is referred to with a negative value, or otherwise a value which is greater than or even equal to the size of the array. It is so because when referring to such a location, we actually refer to a memory location that doesn't exist. A simple aspect as shown in Listing 3.3 can be used to test the array's index if the defined limits are crossed while accessing it. The listed testing aspect assumes that the array is accessed using an index variable at all instances and a global *MAX* is defined.

Listing 3.2: Aging testing using AspectJ-Example II

```
public aspect testMemoryLeakUnreferencedClassB {
        int counter = 0;
        pointcut secclass() : withincode(* ClassA.useB()) &&
            ↪ !within(testMemoryLeakUnreferencedClassB) &&
            ↪ target(ClassB);
        before() : secclass ()
        {
                counter=0; //when classB's object is used
        }
        pointcut priclass() : withincode(* ClassA.useB()) &&
            ↪ !within(testMemoryLeakUnreferencedClassB) &&
            ↪ target(ClassA);
        before() : priclass ()
        {
                counter++; //when classA's object is used
        }
}
```

Listing 3.3: Aging testing using AspectJ-Example III

```
before(int newval) : set(int Array.index) && args(newval)
{
        if (newval>= MAX || newval<0)
                System.out.println("Array out of bound error");
}
```

3.1.2 Concurrency Testing

With the advent of multi-core machines, development of software using multi-threaded design has becoming quite popular. Such design allows maximum utilisation of the available processor cores. Threads that belong to the same process share the resources among themselves. For example, the process memory is shared among all the threads of one process and this makes communication among sharing threads easier and faster. But such ease comes with the cost of possible errors that might surface up due to concurrency.

Testing for concurrent errors like race conditions or deadlocks that may occur while programming with multiple threads using the conventional techniques and tools is not easy. It is so because the nature of threads is non-deterministic and thus the number of possible interleavings is high. Trying all of the interleavings is not practically possible. The probability of producing an interleaving that causes a

concurrent error is low and further reproducing such an interleaving will again be equally difficult [46]. Moreover, the multithreaded programs executed even with the same input values may produce different outputs at different instances [13].

AOP is well suited for concurrency testing. Aspects can be used for increasing the number of different interleavings by noise injection. Noise can be injected into the test execution either randomly or based on some heuristics techniques without making any changes to the source code. The injected noise causes a delay in the execution of thread captured by the pointcut. This in turn gives the other threads, which are ready to run, an opportunity to progress. Thus, this non-invasive noise injection using aspects makes it possible to test the various possible scheduling scenarios of the concurrent programs.

As an example let us consider an application with two threads which operate on a variable. One of the threads divides the value of the variable and the other thread augments this value with the synchronisation condition that the division function must precede the augment function. We have instrumented the *run* method of the division and augment threads with controlled sleep in order to check for concurrent errors that could lead to break of the synchronisation condition. This introduced sleep acts as noise and can be used to test the application as shown in Listing 3.4. In this example listing, we introduce heuristic sleep after the run of *Division* thread with a probability of 1%.

Listing 3.4: Concurrency testing using AspectJ-Example I

```
pointcut noise_Division() : execution(void Division.run());
after() : noise_Division()
{
        try
        {
                if(rand.nextInt(100)==1)
                        Thread.sleep(rand.nextInt(35));
        }
        catch (InterruptedException ex)
        {
                System.err.println(ex.getMessage());
        }
}
```

Similarly, the aspect shown in Listing 3.5 tests the possibility of bugs when shared variables are used in a concurrent application and the synchronisation imposition is neglected by the programmer. It instruments all the accesses to shared vari-

able "a" with heuristic noise. Furthermore, as we shall see in Chapter 5, tester can simply specify basic information regarding the desired test using our Testing Aspect Generator Language (TAGL) which is then automatically converted into the concurrency testing aspect.

Listing 3.5: Concurrency testing using AspectJ-Example II

```
private static Random r = new Random();
pointcut noise_set_a() : set(private int Shared.a);
after() : noise_set_a()
{
        try
        {
                Thread.sleep(r.nextInt(20));
        }
        catch (InterruptedException e)
        {
                e.printStackTrace();
        }
}
```

On the same lines as illustrated above, we instrumented the synchronisation functions like *wait*, *notify* etc. used in concurrent applications with the *call* or *execution* joinpoints and tested for concurrency related errors.

3.1.3 Invariant Testing

An invariant can be defined as a condition or guideline that is mandated to hold true for a program component or may be even for the whole program structure [47]. Testing code for checking of such invariant conditions may lead to scattered code throughout the application under test. We used pointcuts in AOP to capture all the execution points where the invariant condition is supposed to be true and further used suitable advice to check for the correctness of the invariant condition at all such points. This doesn't require any modifications to be made in the source code. Using aspects, invariant conditions imposed at both compile time as well as run time can be tested.

For example, using a simple aspect, we tested the runtime invariant condition that one (or all) methods of a class and all it subclasses should never return a null value as shown in Listing 3.6. The *after returning()* advice captures the return value of all the methods defined by the pointcut and if it is null, an error message is shown.

The method *getSignature()* of the *thisJoinPoint* reference variable returns the signature for the executing join point. Likewise, the method *getSourceLocation()* allows access to the source location information of the joinpoint. We have used the *getSignature()* and *getSourceLocation()* methods in this example to get the signature of methods which return null and the corresponding source code line number where null is returned.

Listing 3.6: Runtime invariant testing using AspectJ-Example I

```
public aspect runtimeinvariant
{
        after () returning (Object obj) : call(Object Classname+.method(..))
        {
                if (null == obj)
                {
                        String error = "Null value returned at " +
                            ↪ thisJoinPoint.getSignature() + " from " +
                            ↪ thisJoinPoint.getSourceLocation();
                        System.out.println(error);
                }
        }
}
```

Compile time invariant conditions can also be checked using AOP. There could be compile time invariant conditions which should hold true for a part of or complete source code like a particular API should never be called or a particular optimised method should be considered for objects of an class or that a private member should not be set outside a *setter* function. AspectJ provides us with *declare warning* and *declare error* which are static crosscutting instructions that we used to generate compile time warnings or errors respectively when a particular usage pattern is detected in the source code of the application. We used the code snippet shown in Listing 3.7 to issue a compile time warning whenever the value of any of the private members of a class, say *Classname*, is set outside the *setter* function.

Listing 3.7: Compile time invariant testing using AspectJ

```
public aspect compiletimeinvariant
{
declare warning : within(Classname) && set(!public * *) && !withincode(*
    ↪ set*(..)) :
   "private field should be accessed only through a setter function" ;
}
```

Similarly, in order to ascertain another runtime invariant condition that a non-static field inside a class in Java should be set only within a constructor and not outside constructor, we used the aspect as shown in Listing 3.8.

Listing 3.8: Runtime invariant testing using AspectJ-Example II

```
public aspect runtimeinvariant
{
        pointcut staticFieldAccess() : set(!static * Classname.*);
        pointcut creation() : execution(Classname.new(..));
        before() : staticFieldAccess() && !cflow(creation())
        {
                throw new Error("non static field should be accessed only
                    ↪ within a constructor");
        }
}
```

3.1.4 Application Programming Interface (API) Testing

API testing is performed on APIs produced by the software development team as well as the third-party APIs. It is performed to check the seamlessness of API calls used in the software. If the APIs used by an application are not tested properly, it can lead to issues not only within the API application but also in the calling application. APIs are tested to verify their return value based on the input condition, successive call to another event/API or any data structure updated by the API. The three key points that need to taken care of while performing API testing are:

- It is important to understand which all API calls need to be tested and how the application in context is using them.
- The input to be provided for testing the API should be identified ensuring that the API's functionality is verified and failures are exposed.
- A suitable tool can be used to generate meaningful inputs for testing and further call the API with these inputs.

The goal of API testing is to verify the correct performance of an external software component before it is integrated with the application. API testing, unlike other

software testing methodologies, primarily focuses on the business logic layer. API testing does not focus on the design and look of the API under test. Whereas we apply inputs from keyboards or files in case of other types of testing, in API testing we need some mechanism to send calls to the API and evaluate its response. Generally a testing tool is deployed which drives the API and also sets up the initial environment when required.

In the context of Java, an API is a collection of pre-written packages, classes, and interfaces which consists of essential methods, fields and constructors. For the purpose of API testing of Java applications, we wrote drivers using aspect and around advice in AspectJ [36, 48] which not only drives the API but also sends the required input parameters. Using aspects, various type of API outputs as listed hereunder were verified:

- An API may return a value based on the input parameters. This situation is comparatively easy to test as the actual output obtained upon test can be compared with the expected output.

- An API might trigger another event, interrupt or API. In this case, the listener of that event, interrupt or API has to be tracked.

- An API may update a data structure or database. It can be verified by accessing the corresponding resource.

As an example, in Listing 3.9 we test the Youtube API for its proper functioning. We first setup the necessary authorisation in the *setup* method and then create an object of the Youtube API within an around advice that executes round the method calling API. Different youtube channel IDs are passed to get their corresponding channel titles which can be tested for correctness as per the known values by implementing the comparison logic in *compareResult* method.

3.1.5 Loop Testing

Loop testing is the test performed to validate the loops in the program for problems related to:

Listing 3.9: Testing the Youtube API

```
public aspect APITest {
        private static YouTube ytObject;
        List<String> range =
            ↪ Lists.newArrayList("https://googleapis.com/auth/youtube");
        Credential crd;

        void setup()
        {
                try {
                        crd = Auth.authorize(range, "APITest");
                }
                catch(IOException e) {
                        System.err.println("IOException: " + e.getMessage());
                        e.printStackTrace();
                }
        }
        void around() : execution(void APICallerClass.APICallerMethod())
        {
                setup();

                //YouTube API Request Object
                ytObject = new
                    ↪ YouTube.Builder(Auth.HTTP_TRANSPORT,
                    ↪ Auth.JSON_FACTORY,crd).setApplicationName
                    ↪ ("APITest").build();

                String Id[] = {"UCB9_VH_CNbbH4GfKu8qh63w",
                    ↪ "UCPDXXXJj9nax0fr0Wfc048g",....};
                for(int i=0; i<Id.length;i++)
                {
                        System.out.print("Title for Channel ID " + i + " is:
                            ↪ " + getChannelTitle(Id[i]));
                        compareResult(i, getChannelTitle(Id[i]));
                }
        }
}
```

- Initialisation: If due to programmer's mistake, there exists uninitialised variables at the beginning of the loop, they are identified when running the loop tests.

- Performance bottlenecks: If loops are increasing the execution time of the application considerably, then it can be determined while loop testing.

- Loop repetition: If a loop is repeatedly executing for excessively high or infinite number of times, it is discerned with loop testing and can be fixed up.

Loop testing can be done, for example, by trying to have a loop executed with a fewer than minimum, as well as a larger than maximal number of iterations. In general, a simple loop is tested in the following manner:

- Skip the execution of the loop completely
- Only 1 pass through the loop
- Only 2 passes through the loop
- "x<n" passes through the loop where n the maximum number of allowed passes through the loop
- n, n-1, n+1 passes through the loop

We propose that AOP can be used for carrying out loop testing. For example, aspects in AspectMatlab can be deployed for testing Matlab applications where the loops are extensively used. AspectMatlab provides us with the ability to capture the loops through a range of pointcuts namely: *loop, loopbody and loophead* [49]. The scope of these three pointcuts has been depicted in Figure 3.2 and explained in Table 3.1.

Since loops cannot be named in Matlab, therefore loop iterator variables are used to capture a loop pattern. Further since the names for loops iterator variables used in source code are often quite general (usually i or j), therefore we used the *within* pattern to restrict the scope of matching to desired and meaningful joinpoints. For example, pointcut *loopsMyClass : loop(i) & within(class, myClass)* can be used to capture all the loops written in the class *myClass* which iterate over the iterator *i*. Likewise, the pointcut *patternMethodCallInsideLoops : call(func) & within(loops, *)* can be used to capture every call to the *func* method inside any of the loops in the program.

For testing loops in Matlab, we used the AspectMatlab's *loopiterator* construct to replace or modify the value held by the loop iterator variable and assign desirable

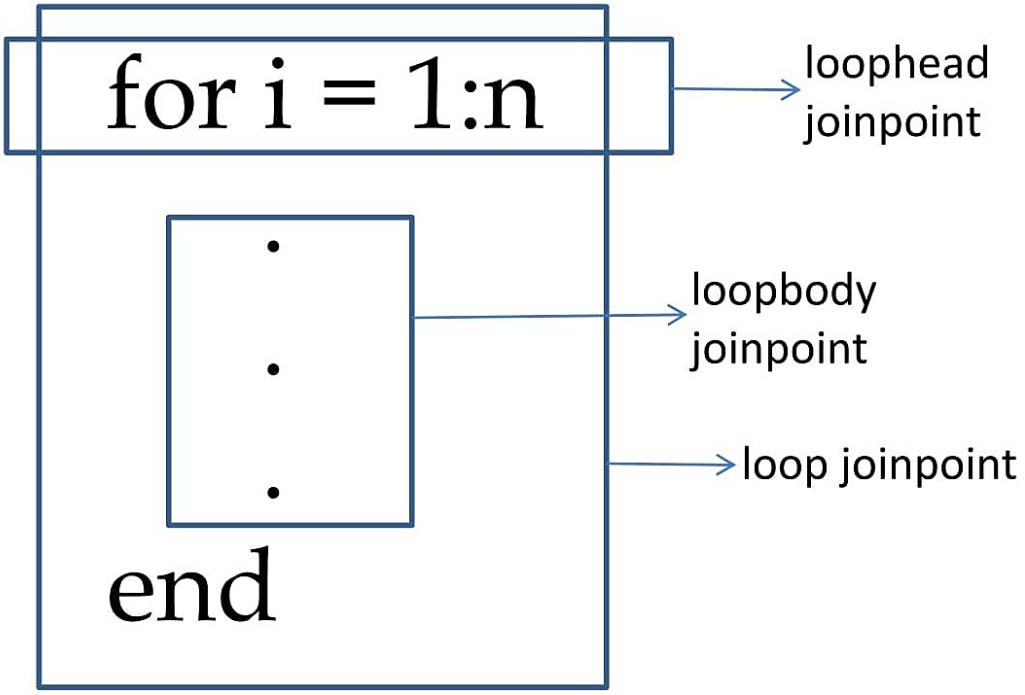

Figure 3.2: AspectMatlab Loop Joinpoints

Table 3.1: AspectMatlab Loop Pointcuts

Pattern	**Crosscuts**
loop(i)	Captures execution of loops which iterate over i
loophead(j)	Captures the header of all loops that iterate over j
loopbody(*)	Captures the body of every loop in the program

values for testing. For example, the aspect snippet shown in Listing 3.10 captures all the loops that iterate over *i* and are within the class *myClass*. The construct *args* collects the values of loop iterator variable *i*. The captured loops are then made to iterate with a halved value of the iterator using the *body* call which is similar to the *proceed* call in AspectJ and is used to execute that portion of code which corresponds to the body of the loop.

Although AspectJ does not provide any pointcut patterns to capture the loop joinpoints directly [18], still we could use it for loop testing by changing the value of the loop variable using appropriate pointcuts within aspects and then make it run one or more times, as desirable. Listing 3.11 shows example of an aspect which captures the loop iterator variable *i* and makes the loop inside the *loopRunMethod* function of the class *myClass* run only once. The value of *iterationsRequired* in this aspect can be adjusted to change the number of runs of the loop. The loop

Listing 3.10: Loop testing with AspectMatlab

```
patterns
        loopsMyClass : loop(i) & within(class, myClass);
end
actions
         halfiterator  : around loopsMyClass : (args)
                  for loopiterator = args/2
                           body() %Run the loop with halved value of i
                  end
         end
end
```

testing aspect shown in Listing 3.11 assumes that the loop iterator variable is declared as a class member.

3.1.6 Basis Path Testing

Exhaustive testing can be carried out by testing each and every path through every module at least once, but this is not impractical from the point of view of time taken and enormous number of possible paths. Thus different strategies exist to select the paths of an application for testing. Basis path testing is one such technique that is based on the cyclomatic complexity to determine the number of independent paths.

Basis path testing is the oldest white box testing technique first proposed by Tom McCabe [50] that uses the control flow of the program to design test cases. The code is converted into a control flow graph model which is used to derive the independent test paths. Each possible linearly independent path in the program is then tested for correctness. A basis path is a unique path through the software in which there are no iterations. When all of these paths have been executed, we can be sure that every statement in the code (under test) has been executed at least once and that every branch has been exercised for true and false conditions.

In essence, basis path testing involves the following 4 steps:

1. Compute the flow graph G
2. Calculate the cyclomatic complexity V(G)
3. Select a basis set of independent paths

Listing 3.11: Loop testing with AspectJ

```
public class myClass {
        int i;
        void loopRunMethod()
        {
                for(i=0; i<100; i=i+1)
                {
                        //Java statements to be executed
                }
        }
}
public aspect LoopTest {
        static int j = 0;
        int iterationsRequired = 1;
        Object around(int i) : set(int myClass.i) && !within(LoopTest) &&
            ↪ args(i)
        {
                if(j<iterationsRequired)
                {
                        i = 0;
                        j++;
                }
                else
                {
                        i = 100;
                }
                return proceed(i);
        }
}
```

4. Generate test cases for each of these paths and execute these to find bugs

For the second step above, the cyclomatic complexity is calculated using the below formula:

$$V(G) = e - n + p \tag{3.1}$$

where e is the number of edges, n is the number of vertex and p is the number of connected areas in the graph G.

As basis path testing involves finding all possible independent test paths through the source code and exercising and testing each path at least once, we propose that a tracing aspect [18] can be written which finds out all the possible execution paths in a program and further another aspect can be used to execute selected paths

using different input parameters. Pointcuts with wild cards in AOP can be used to select the desired (one or more) execution points and perform the basis path testing. For example, the pointcut *** Account+.*(..)** captures all the methods in the Account class as well as its subclasses. This will also match any new method that has been introduced in the Account class' subclasses.

3.2 Black Box Testing

Black box testing is the counterpart of white box testing. We need to have combination of both black box and white box testing techniques to cover maximum bugs in the application. Black box testing is called so because therein we test without having knowledge about the internal structure, the process undertaken or any other internal aspect of the system under test. For example, we test a search engine simply by providing different values for the search query text without worrying about how the search engine algorithm internally fetches the resultant web links. Black box testing is thus mainly focused on the functionality of the system under test and the results. The bugs found using black box testing are generally bugs related to functionality, validation or Graphical User Interface (GUI). The mechanism of black box testing is depicted in Figure 3.3.

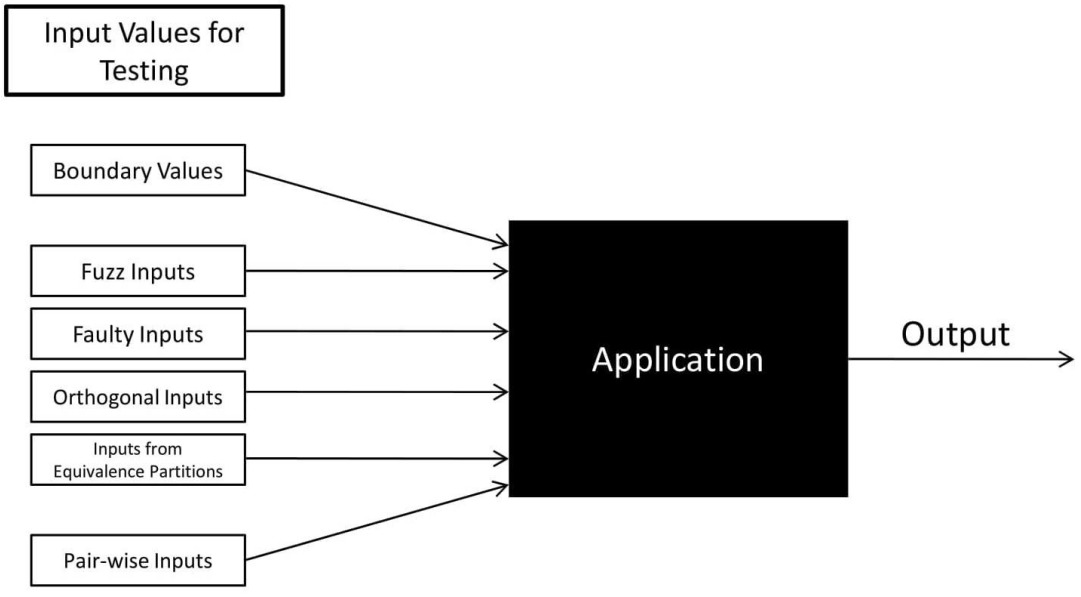

Figure 3.3: Black box testing

We performed black box testing using AOP as shown in Listing 3.12. In this AspectJ example, the around advice replaces the execution of the function that

matches the defined pointcut viz. *ClassName.Function*. An array of inputs is created and then *proceed* is called which executes the matched function with these inputs one by one. The output obtained is captured and further compared with the expected output as per software requirement specification (SRS). The function *compareResult* written inside the aspect compares results obtained from every input with the expected outputs.

Listing 3.12: Black box testing using AspectJ

```
int around(int origArg) : call(int ClassName.Function(int)) &&
    ↪ args(origArg)
{
        int[ ] testInput = new int[ ] {−1,0,1,100,10000,2147483647};
        for (int i = 0; i < testInput.length; i++)
        {
        int actualResult = proceed(testInput[i]); //invoke test
        compareResult(testInput[i], actualResult);
        }
        return proceed(origArg); //proceed with original arguments
}

void compareResult(int input, int actualResult)
{
        //implemented as per the specifications laid down in the SRS
}
```

Similarly the source code in Listing 3.13 shows how we can use Aspect Python to perform the black box testing of a method in python class that reads and returns the content of a file. Upon testing, we observed that although the tested method passed the test and generated an exception when the argument file did not exist but it could not handle the case when null (*None* in python) was passed for the file name.

3.2.1 Boundary Value Testing

Boundary value testing is a type of black box testing which is performed to check the defects at the boundary conditions. It is based on the idea that the errors usually occurs at the boundary values of inputs because the programmer fails to cater the special processing required for such boundary values. We used aspects to specify the boundary values test cases and further execute such test cases and compare the results in a manner similar to that shown in Listing 3.12. In case if

Listing 3.13: Black box testing using Aspect Python

```
import aspectlib

@aspectlib.Aspect
def strip_return_value(*k):
        #Test Case 1: File exists
        content1 = yield aspectlib.Proceed("filename.txt")
        print(content1)
        #Test Case 2: File does not exists
        content2 = yield aspectlib.Proceed("/*.txt")
        print(content2)
        #Test Case 3: Null value passed
        content3 = yield aspectlib.Proceed(None)
        print(content3)

class file_read:
        @strip_return_value
        def read(name):
                try:
                        fp = open(name,'r')
                        return fp.read()
                except IOError:
                        print("File could not be opened")
```

(two or more) variables with input domains are present, then the boundary value test cases are determined by selecting nominal value for one of the variables and then selecting minimum, slightly above the minimum, maximum, slightly below the maximum and one nominal value for the other variable as enlisted in Table 3.2.

Table 3.2: Boundary value test cases with two variables X and Y

X Value	**Y Value**
X_{nom}	Y_{min}
X_{nom}	Y_{min+}
X_{nom}	Y_{max}
X_{nom}	Y_{max-}
X_{min}	Y_{nom}
X_{min+}	Y_{nom}
X_{max}	Y_{nom}
X_{max-}	Y_{nom}
X_{nom}	Y_{nom}

Listing 3.14 shows an example for the boundary value testing of a function which

takes date, month, year (in the 100 year range of 1917 to 2017) as input and returns the corresponding day.

Listing 3.14: Boundary value testing using AspectJ

```
public aspect boundaryValueTestingAspect {
        String around(int date,int month,int year) : execution(String
            ↪ DateToDay.returnDay(int,int,int)) && args(date,month,year)
        {
                int[] dateTestValues = {15,15,15,15,15,15,15,15,15,1,2,30,31};
                int[] monthTestValues = {6,6,6,6,6,1,2,11,12,6,6,6,6};
                int[] yearTestValues =
                    ↪ (1917,1918,1967,2016,2017,1967,1967,1967,1967,1967,
                    ↪ 1967,1967,1967};
                int i=0;
                String actualDay="";
                for (i=0;i<dateTestValues.length;i++)
                {
                        actualDay = proceed(dateTestValues[i],
                            ↪ monthTestValues[i],yearTestValues[i]);
                        compareResult(dateTestValues[i],monthTestValues[i],
                            ↪ yearTestValues[i], actualDay);
                }
                return proceed(date,month,year);
        }

        void compareResult(int date,int month,int year,String actualDay)
        {
                //as per the specification of the function returnDay
        }
}
```

As we shall see in Chapter 5 later, tester can simply specify the range of values for the parameters and the expected values using our Testing Aspect Generator Language (TAGL) code which is then automatically converted into aspects with boundary value test cases (test cases with values on the boundaries of input domain and values just above and below the extreme edges of input domain) and an around advice to compare the results.

3.2.2 All Pairs Testing

It is impractical to test for all the possible combinations of values for all the parameters of a method. For example, if there is a page in an application with

a listbox which can take any of the listed ten values, a text box which can take numerical values between 1 to 100 and further a checkbox which can be either checked or unchecked. Now to test such a page for its proper functionality, the total number of possible test cases would be $10 \times 100 \times 2 = 2000$, but it would be practically cumbersome and time taking to test for all of these. In fact, this number shall go further high if we consider the negative or invalid inputs as well for the purpose of testing.

The idea behind all pairs testing, which is also called pairwise testing, is that it is enough to test using combinatorial method wherein we test with all the possible discrete combination of the involved interacting parameters. The values of variables in the test cases are permuted to achieve coverage of all the possible pairs and thus reducing the number of tests to perform. Pairwise-generated test cases identify all the pair combinations and cover all such combinations of two. Pairwise testing is basically based on the principle of coupling effect which suggests that if there is a fault that manifests with a specific setting of configuration variables, it is most likely caused actually by only a small subset of those variable values.

Table 3.3 lists the possible values for the three arguments of the *sanctionLoan* method of the class *loanClass* which determines whether loan can be sanctioned or not based on the borrower's characteristics, namely number of kids, occupation and borrower's loan history. Listing 3.15 shows how an aspect can be used to perform all pairs testing of this method. The around advice executes the pairwise test cases and then calls the *compareResult* method to compare the obtained results with the expected results. Further, using our TAGL (as explained later in Chapter 5), the tester just needs to specify the various possible values for the involved parameters and the generated aspects shall automatically produce the test cases based on the all pairs testing algorithm.

Table 3.3: All Pairs Testing: Variables and their possible values

Variable name	**Possible values**
Number of Kids (no_of_kids)	2, 3, 4
Occupation (occupation)	Job, Business
Loan History (firstloan)	True, False

Listing 3.15: All pairs testing using AspectJ

```
public aspect allPairsTestingAspect {
        boolean around(int no_of_kids,String occupation,boolean firstloan)
            ↪ : execution(boolean loanClass.sanctionLoan(int,String,
            ↪ boolean)) && args(no_of_kids,occupation,firstloan)
        {
                int[] no_of_kids_Array = {2, 2, 3, 3, 4, 4};
                String[] occupationArray = {"Job", "Business", "Job", "
                    ↪ Business", "Job", "Business"};
                boolean[] firstLoanArray = {true, false, false, true, true,
                    ↪ false};
                int i=0;
                boolean decision;
                for (i=0;i<no_of_kids_Array.length;i++)
                {
                        decision = proceed(no_of_kids_Array[i],
                            ↪ occupationArray[i],firstLoanArray[i]);
                        compareResult(no_of_kids_Array[i],occupationArray[i
                            ↪ ],firstLoanArray[i], decision);
                }
                return proceed(no_of_kids,occupation,firstloan);
        }

        void compareResult(int no_of_kids,String occupation,boolean
            ↪ firstloan,boolean decision)
        {
                //as per the specification of the function sanctionLoan
        }
}
```

3.2.3 Orthogonal Testing

Orthogonal testing is based on the fact that interactions are a major source of defect and that most of the defects arise from simple interactions. Orthogonal testing involves selecting input combinations using orthogonal array technique which guarantees pairwise coverage of all variables. The test set created using orthogonal array technique is concise with fewer test cases as compared to testing with all possible combinations of variables. For example, lets suppose that we have a system where we have four variables - lets say temperature, pressure, humidity and rainfall and each of which can take three values as shown in Table 3.4. Now the total number of test cases in the exhaustive test set shall be 81 (3 X 3 X 3 X 3). However, the test set created using the orthogonal array technique shall have

only 9 test cases as explained below.

An orthogonal array is represented by the following three elements:

- Runs (N) - Number of rows in the array, which represents the number of test cases.
- Factors (K) - Number of columns in the array, which correlates to the maximum number of variables under consideration.
- Levels (V) - Maximum number of values that can be taken by any single variable.

A typical representation of an orthogonal array is $L_{Runs}(Levels^{Factors})$. For the above example with four variables having three values, the orthogonal array shall be $L_9(3^4)$. Table 3.5 shows a standard orthogonal array with four factors and three levels:

Table 3.4: Orthogonal Testing: Variables and their possible values

Variable name	**Possible values**
Temperature	100C, 150C, 200C
Pressure	2psi, 5psi, 8psi
Humidity	Absolute, Relative, Specific
Rainfall	Low, Moderate, Heavy

Table 3.5: Standard Orthogonal Array $L_9(3^4)$

Exp. No.	**Factor A**	**Factor B**	**Factor C**	**Factor D**
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Now in order to generate the orthogonal test cases, we can map the factors and values in Table 3.5 with the actual variables and their values as shown in Table 3.4.

For our example, the test cases so generated are shown in Table 3.6. As evident from this table, the total number of test cases shall be only 9. In the orthogonal array, the permutations of various factors and their levels are chosen in such a way that the responses obtained are not correlated. Therefore, each test case generated using this technique renders a unique piece of information and fewer test cases are sufficient to catch the fault.

Table 3.6: Orthogonal Test Cases $L_9(3^4)$

Test Case No.	**Factor A**	**Factor B**	**Factor C**	**Factor D**
1	100C	2psi	Absolute	Low
2	100C	5psi	Relative	Moderate
3	100C	8psi	Specific	Heavy
4	150C	2psi	Relative	Heavy
5	150C	5psi	Specific	Low
6	150C	8psi	Absolute	Moderate
7	200C	2psi	Specific	Moderate
8	200C	5psi	Absolute	Heavy
9	200C	8psi	Relative	Low

We performed black box testing with orthogonal test cases by using aspects and results were compared in a way similar to explained in Section 3.2.2. Moreover, as we shall see later in Chapter 5, the tester can simply specify the various possible values for the variables under test in the form of TAGL statements and then testing aspect with appropriate test cases as per the orthogonal array technique are automatically generated.

3.2.4 Fuzz Testing

Fuzz testing is to test using massive unexpected or random data as inputs, for example test inputs like 31/2/2014 or long strings etc. Based on the process of generation of inputs, fuzzing can be mutation-based (modify parts of the known valid input files or arguments) or generation-based (generate malformed files or arguments automatically based on known formats). Fuzzing aims to discover failures caused by malformed insensible values that the programmer might have missed to code for. Fuzz testing monitors the effect of such inputs on the system under test, specifically for system crashes, hangs or any other abnormal reaction.

We used AOP to perform fuzz testing. Aspects were used to generate random inputs and further compare the obtained results with the expected results. There

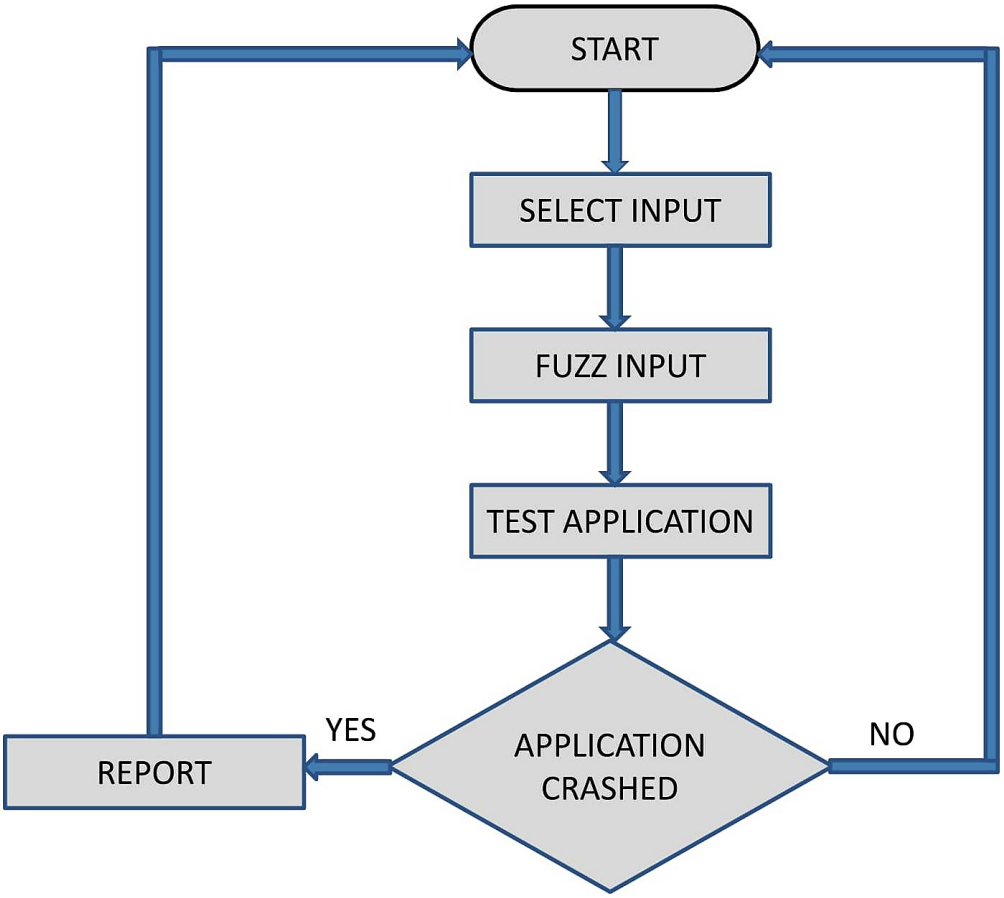

Figure 3.4: Flow chart for fuzz testing

could be commonly used libraries or files of different formats processed by the target software which can be fuzzed to test the system for robustness. For example, for a function in an application which reads input from a text file, one sample test case could be "check that if the file content is fuzzed or modified by an attacker, then is the application robust enough to handle/detect such an alteration". We used aspects with a before advice to fuzz the content of an input file with random data before its contents are read and further test whether the application is able to detect the alteration or not. File fuzzing aspect shown in Listing 3.16 substantiates our point.

The input generated for file fuzzing should be semi-valid or in other words, it should be valid enough such that it is not detected by the application and at the same time erroneous enough to cause the application to fail. If the application fails, then such fuzzing input is saved, bug reported and the following iteration with next fuzz inputs is started as shown in the flow chart in Figure 3.4. For

Listing 3.16: Fuzz testing using AspectJ

```
public aspect FileWriteAspect {
        pointcut readFile() : execution(void FileOperation.ReadFile());
        before() : readFile()
        {
                //The path of the file to fuzz
                String fileName =
                    ↪ "/home/administrator/inputdir/filename.txt";
                FileReader fileReader = new FileReader(fileName);
                //Character array to read data from file
                char[] buffer = new char[1000];
                BufferedReader br = new BufferedReader(fileReader);
                br.read(buffer);
                br.close();
                //Unusual characters array, supplied based on tester's
                    ↪ experience and application's context
                char[] fuzzData = {'\', '%', '0', '1', '\', '%', '0', '2',
                    ↪ '\', '%', '0', '3', '\', '%', '0', '4'};
                //Randomize the above array
                Random random = new SecureRandom();
                char[] fuzz = new char[array.length];
                for (int i = 0; i < array.length; i++)
                {
                        fuzz[i] = array[random.nextInt(array.length)];
                }
                System.arraycopy(fuzz, 0, buffer, 5, fuzz.length);
                FileWriter fileWriter = new FileWriter(fileName);
                BufferedWriter bw = new BufferedWriter(fileWriter);
                //Write the modified buffer array into the file
                bw.write(buffer);
                bw.close();
        }
}
```

file fuzzing, there could be three options, namely: *insert, overwrite, replace* [51]. In case of *insert*, the fuzz input is inserted before or after a specified field of the input file as shown in Figure 3.5. In case of *overwrite*, the fuzz input overwrites a part of a specified field of the file as shown in Figure 3.6. In case of *replace*, the field specified by the tester is completely replaced by the fuzz input as shown in Figure 3.7. The aspect shown in Listing 3.16 is an example of *overwrite* where the content of the input file starting from position 5 is overwritten by the fuzz input.

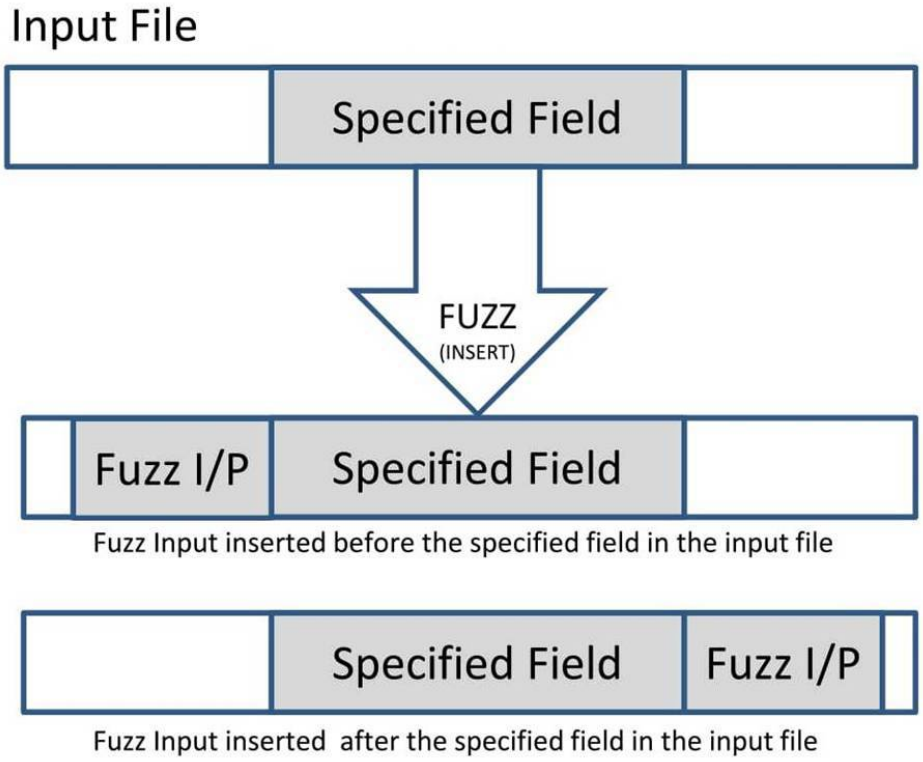

Figure 3.5: File fuzzing: Insert fuzz values into the input file

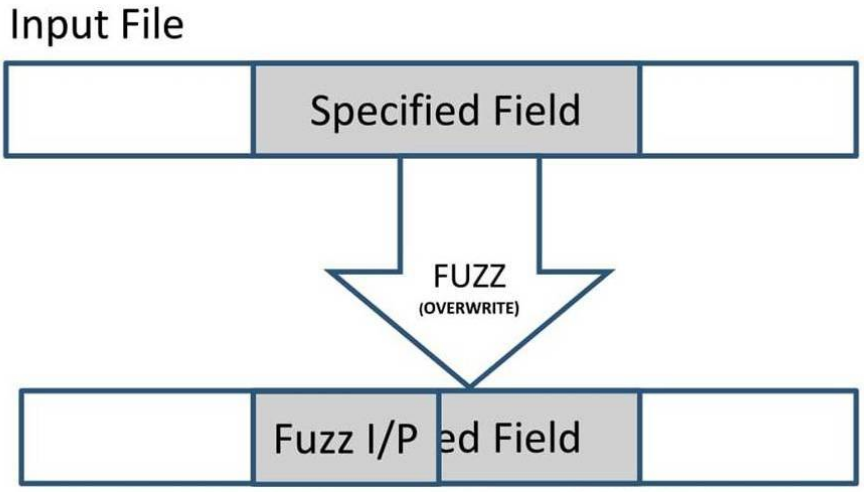

Figure 3.6: File fuzzing: Overwrite a specified field of the input file

3.2.5 Fault Injection Testing

Fault injection testing focuses on the error handling capabilities of a software. Fault injection testing helps to test certain code paths which shall otherwise be rarely executed when testing with the routine cases. In order to test for the error handling capability, we need to generate test cases with erroneous inputs and some amount of code instrumentation is always necessary [52]. Suitable aspects can be

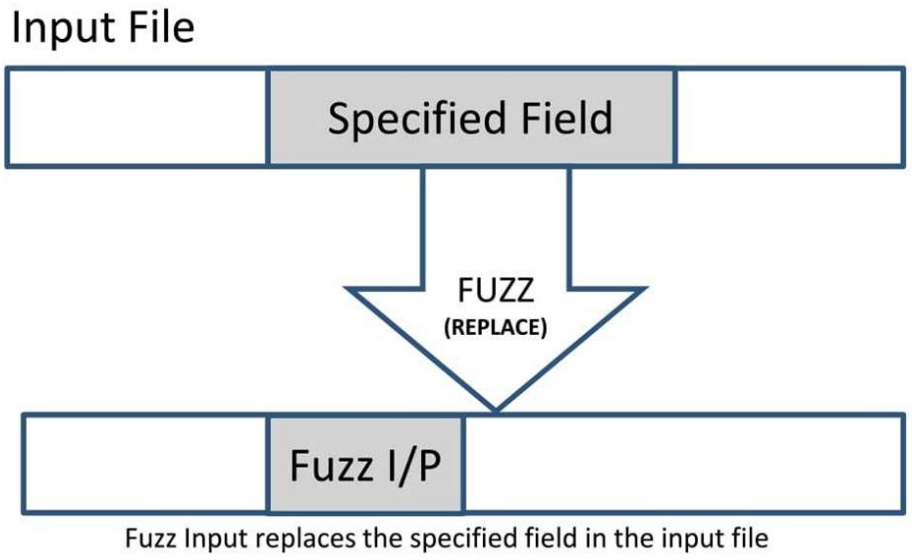

Figure 3.7: File fuzzing: Replace a specified field of the input file

written for this purpose for e.g. we used aspects to produce a representative set of transactions that contains errors or to bring about corruption in the network packets or to make an assertion false during program execution. Using pointcuts, we inserted such faults at desirable execution points. For example, the pointcut shown in Listing 3.17 can be used to test the behaviour of a banking application upon inserting the fault of depositing or withdrawing *null* amount. The *after throwing* advice shown in listing is meant to collect the context of the program where the error occurs and the method throws an exception.

3.2.6 Equivalence Partitioning Testing

Equivalence partitioning is yet another type of black box testing in which the test data is classified into equivalence classes. In case of equivalence class partitioning, the domain of input values is partitioned in such a way that the behaviour of the application is same for every value belonging to the same equivalence class. For example, for a method that takes as input the candidate's annual income and calculates the tax, there are various income tax slabs and we can define different equivalence classes of input values for income based on the tax slabs. We used aspects to define test cases from the partitions of equivalent data and further execute these test cases and compare the results in a manner similar to that shown in Listing 3.12. Moreover, as we shall see later in Chapter 5, tester can specify the

Listing 3.17: Fault injection testing using AspectJ

```
public aspect faultInjectionAspect {
        void around(Long amount) : (execution(*
            ↪ BankingClass.depositMethod(Long)) || execution(*
            ↪ BankingClass.withdrawMethod(Long))) && args(amount)
        {
                amount = null;
                proceed(amount);
        }

        after() throwing(Throwable ex) : (execution(*
            ↪ BankingClass.depositMethod(Long)) || execution(*
            ↪ BankingClass.withdrawMethod(Long)))
        {
                Signature sig = thisJoinPointStaticPart.getSignature();
                System.err.println(sig.getDeclaringType().getName() + "." +
                    ↪ sig.getName() + "LOC: " +
                    ↪ thisJoinPoint.getSourceLocation());
        }
}
```

range of input parameters in the form of TAGL statements and then aspects with appropriate test cases from equivalence partitions are automatically generated.

3.3 Non Functional Testing

Although existing testing techniques are worthwhile for functional testing, but the non functional requirements which are increasingly surfacing up in recent software applications set urge for new effective methodologies for testing. Non functional testing is about verifying the proper implementation of the non-functional requirements of an application like performance, robustness, recovery etc. The implementation of these non-functional requirement is mostly crosscutting in nature and therefore testing these with conventional techniques shall require heavy instrumentation of the original source code at various places. Since AOP addresses the issues arising from crosscutting concerns well, we propose its use for testing the non-functional requirements of software applications.

Based on the non-functional requirements as per the software requirement specification (SRS), the testing objectives are formulated and grouped into different categories on the basis of their characteristics and the non-functional concern they

correspond to. We modularised the testing of such crosscutting non-functional concerns by capturing them using testing aspects. For example:

- To carry out performance testing of an application, we used aspects to find out the memory usage or measure execution times and assessed the quality of the implementation.

- To test the system for robustness under variety of failure conditions, we used the aspects to create situations of starvation like sudden disconnection from network.

- To test the system reliability under specified conditions for a specified period of time, we used aspects to generate repeated inputs and execute for a designated period of time to find out how long the software will execute without failure.

- To perform penetration testing in order to test the software for safety against intruders, we used aspects to gather information about the target program before the test, identify possible entry points, attempt to break in and reporting back the findings.

- For monitoring purpose, we used a monitor aspect which can, for example, capture dangling null-pointers anywhere in the software system like unexpected calling of a method by a null object or creation of an object without initialisation.

The example in Listing 3.18 shows how we implemented aspects for measuring the execution times of various methods of a class which can be used to test the performance requirements.

Likewise, aspect shown in Listing 3.19 can be used to measure the memory usage of the various methods of a class, say *Student*.

The example in Listing 3.20 shows how we implemented aspects for monitoring call to any method by a null object anywhere within a class.

In the following subsections, the two most important types of non functional testing, namely load testing and security testing are discussed and usefulness of aspects to carry out these testing is presented.

Listing 3.18: Measure execution times using AspectJ

```
public aspect findExecutionTimeAspect
{
        pointcut captureAllMethodCalls(Object object) : call(*
            ↪ Classname.*(..)) && target(object);
        before(Object object): captureAllMethodCalls(object)
        {
                System.out.println("Method called using " + object);
        }

        Object around(Object object) : captureAllMethodCalls(object)
        {
                System.out.println("Method " + thisJoinPoint + " started at "
                    ↪ + System.currentTimeMillis());
                proceed(object);
                System.out.println("Method " + thisJoinPoint + " completed
                    ↪ at " + System.currentTimeMillis());
        }
}
```

Listing 3.19: Measure memory usage using AspectJ

```
public aspect memoryUseAspect {
        pointcut captureAllMethodCalls(Object object) : execution(*
            ↪ Student.*(..)) && target(object);

        before(Object object): captureAllMethodCalls(object)
        {
                System.out.println("Method called using " + object);
        }

        Object around(Object object) : captureAllMethodCalls(object)
        {
                long memStart = Runtime.getRuntime().totalMemory() -
                    ↪ Runtime.getRuntime().freeMemory();
                Object retObject = proceed(object);
                long memEnd = Runtime.getRuntime().totalMemory() -
                    ↪ Runtime.getRuntime().freeMemory();
                long memUse = memStart - memEnd;
                System.out.println("Memory usage by Method: " +
                    ↪ thisJoinPoint + "is: " + memUse);
                return retObject;
        }
}
```

Listing 3.20: Aspect to monitor method call by a null object

```
public aspect monitorAspect
{
        pointcut nullPointerTest(Classname obj) : call(* Classname.*(..)) &&
            ↪ target(obj);
        before(Classname obj) : nullPointerTest(obj)
        {
          if(obj==null)
                        System.out.println("Null pointer exception in: " +
                            ↪ thisJoinPoint + " at: " +
                            ↪ thisJoinPoint.getSourceLocation());
        }
}
```

3.3.1 Load Testing

This type of non-functional testing tests the behaviour of a software application under normal and peak input loads. The main goal of load testing is to examine the extremum of application in terms of database, network, hardware etc. It helps to identify the bottlenecks in the system and database components under various workloads which should be rectified before the system is put to real use. Load testing is generally performed by creating virtual and distinct users that emulate workload for the system under test.

We used aspects in AOP to generate the load required for testing. For example, in order to test the performance of a shopping cart application, an aspect to create multiple cart users all of whom performed the shopping operation concurrently was written. The simple aspect code snippet shown in Listing 3.21 demonstrates the same. In this aspect, one thousand cart users are created using the *addUser* method of the *shopping* class.

As another example, a single user of the cart can be made to purchase multiple items to test how well the application handles large purchase values as shown in Listing 3.22.

Moreover, ramp testing, which is a strategy of load testing in which we check the performance of the software with constantly increasing load, is also possible using aspects. For example, the shopping cart application can be ramp tested using an aspect which increases the numbers of cart users over a given period of time. This way we can determine the maximum number of users the application can sustain before it starts producing error messages. We performed the testing of popular

Listing 3.21: Load testing of a shopping cart application-I

```
public aspect loadTestShoppingCartI {
        pointcut testShoppingCart(shopping s) : call(void
            ↪ shopping.addUser(User)) && target(s) &&
            ↪ !within(loadTestShoppingCartI);
        void around(shopping s) : testShoppingCart(s)
        {
                User[] userArray = new User[1000]; //Create 1000 new
                    ↪ dummy users
                for(int j=0;j<1000;j++)
                {
                        userArray[j] = new User();
                        s.addUser(userArray[j]);
                }
        }
}
```

Listing 3.22: Load testing of a shopping cart application-II

```
public aspect loadTestShoppingCartII {
        int around(User u) : call(int User.shop(Item)) && target(u) &&
            ↪ !within(loadTestShoppingCartII)
        {
                Item[] itemArray = new Item[5000]; //Purchase 5000 items
                int shopAspectValue = 0;
                for(int j=0;j<5000;j++)
                {
                        itemArray[j] = new Item();
                        shopAspectValue = u.shop(itemArray[j]);
                }
                return shopAspectValue;
        }
}
```

Java chatting application *NetC* using this mechanism and plotted a graph for the application's behaviour having axes of number of users vs. RAM usage. This shall be discussed in detail in Section 4.1 of Chapter 4.

3.3.2 Security Testing

Security testing is performed to test the authorisation mechanism of the software and evaluate the provisions of software against attacks. AOP allows the security testers to develop and inject separate modules within aspects for conducting security testing of the applications, independent of their business logic [53]. Moreover, security is a cross cutting concern and thus code to implement this concern shall be spread all over the program. AOP, by its definition, handles cross cutting concern well and thus is most suitable for security testing while avoiding the issue of test code scattering.

In the context of Java, there are Servlets or Java Server Page (JSP) applications which take user parameter in the form of strings and render Hypertext Markup Language (HTML) pages with these string inputs. The attacker can create Uniform Resource Locator (URL) with malicious javascript code as input which shall be written into the HTML response page. If a user is made to click on such a URL, the malicious javascript code shall be executed in the user's browser. Malicious script code can be written to access client's cookies, redirect the client to harmful sites, or even for session hijacking.

In our work, we used aspects for the purpose of testing Java Servlets. Java servlets are used to create web applications with dynamic content and they reside on the web server. As servlets exist on world wide web, these are prone to various security attacks like resource tampering, denial of service attack, structured query language (SQL) code injection etc. The servlets need to be tested for existence of vulnerabilities which may arise due to unawareness of or mistake left by the programmer. Thus, it is necessary to perform security testing of the servlets with proper test cases.

A servlet container like Apache Tomcat maps the Hyper Text Transfer protocol (HTTP) request to the corresponding servlet. For this purpose, the container creates light-weight threads for handling the multiple client requests to a single instance of the servlet. The client browser uses two methods, namely GET Method and POST Method, to pass user information to the web server. Accordingly,

the servlet handles the client requests with information using the *doGet()* or the *doPost()* method.

We propose the use of aspects which implement the *Filter* interface to capture desirable execution points within the Java servlets and perform their security testing. We first create a *RequestWrapper* class that extends the *HttpServletRequestWrapper* class of the Java Servlet package. Within this *RequestWrapper* class, we override the *getParameter()* method to pass parameters for the purpose of security testing to the Servlet. Listing 3.23 shows how the *RequestWrapper* class is written extending the *HttpServletRequestWrapper*.

Listing 3.23: RequestWrapper class for servlet testing

```
public class RequestWrapper extends HttpServletRequestWrapper
{
        public RequestWrapper(final ServletRequest request)
        {
                super((HttpServletRequest) request);
        }

        //The getParameter method is overridden here
        @Override
        public String getParameter(final String name)
        {
                //Write Code for passing testing parameters
                 ... ... ...
        }
}
```

Thereafter, we used the aspect shown in Listing 3.24 to implement the *Filter* interface and create an object of the *RequestWrapper* class within the *doFilter()* method.

The aspect shown in Listing 3.24 along with the *RequestWrapper* class can be used to pass different parameters to the Servlet and thereby testing it for possible vulnerabilities. For example, to test the servlet for SQL injection attack, we can pass malicious SQL commands using the *getParameter()* method in the *RequestWrapper* class. Further the test results obtained by passing different parameters or the exceptions so generated can also be monitored using aspects.

As another example of security testing using aspects, we propose that aspects can also be used to test the access control mechanism intended in an application. For example, if it is intended in a banking application that the methods of the

Listing 3.24: Aspect for servlet testing

```
public aspect testAspect implements Filter
{
        public void doFilter(ServletRequest request,ServletResponse
            ↪ response,FilterChain chain) throws
            ↪ IOException,ServletException
        {
                //RequestWrapper class constructor called
                chain.doFilter(new
                    ↪ RequestWrapper((HttpServletRequest)request),response);
        }
        @Override
        public void destroy()
        {
                //Necessary to implement
        }
        @Override
        public void init(FilterConfig arg0) throws ServletException
        {
                //Necessary to implement
        }
}
```

BankAccount class should be accessible only from the *BankAccountHolder* class, then the aspect shown in Listing 3.25 can be used to test such access control.

Listing 3.25: Aspect for testing access control

```
public aspect BankAccountAccessControlTest
{
        declare error : (call(* BankAccount.deposit*(..)) || call(*
            ↪ BankAccount.withdraw*(..)) || call(*
            ↪ BankAccount.transfer*(..))) && !within(BankAccountHolder) :
            ↪ "Unintended access to BankAccount";
}
```

3.4 Testing at different levels of the software development process

Each phase of the software development life cycle goes through testing and thus there exists various levels of testing [54]. Unit level testing is done to test the

individual components or modules. Integration level testing is done to evaluate the effect of one module over the other and involves testing software components in combination with other components. Acceptance testing is carried out to verify that the software meets the customer specified requirements. We shall discuss the usefulness of AOP to perform these software testing types which are performed at different levels of the software development process in the following subsections.

3.4.1 Unit Testing

In unit testing, the smallest units of code are tested independently so that whenever an error is detected its cause can be identified and isolated to one particular method or class. As the unit testing is typically performed on every method of every class in the target program, therefore it is a cross cutting concern and the testing code using conventional techniques shall be scattered. AOP, by its definition, can be used to modularise this cross cutting concern by writing the testing code within aspects [48]. We performed several unit tests on independent program units using white box and black box strategies with the help of aspects.

JUnit is the most popular Java automated testing tool [55] which is largely used by the testers for unit testing. We mapped the testing annotations in JUnit, namely *@Test*, *@Before*, *@BeforeClass*, *@After*, *@AfterClass* onto the *around, before, after* and *adviceexecution* advices available in AspectJ. The pre-conditions can be setup in the before advice, the reset or release of resources can be performed in the after advice and the method to be tested can be instrumented and tested with desired input values using the around advice. Regarding this analogy, that we utilised for performing unit testing with aspects, we shall discuss in detail in Chapter 6 wherein we have compared our proposed technique with the conventional testing techniques.

Moreover, when a private method contains an algorithm which requires more unit testing than it is possible through the public interfaces, then it becomes necessary to directly test the operations of the private method as well. Using AspectJ, the private methods can be easily accessed and tested in the testing aspect by declaring the testing aspect as *privileged*. Code inside privileged aspects has access to all members of the captured object, even the private ones.

3.4.2 Integration Testing

When separate modules of a software are integrated together and tested as a group for issues like data communication, defects in interfaces etc., it is called Integration Testing. The prime motive behind Integration Testing is to discover inconsistencies arising due to shared data areas and inter process communication between the various modules. Integration testing of large applications comprises of combining many modules together which are tightly coupled with each other.

Integration testing is not conducted at the end of the cycle, rather it is conducted simultaneously with the development process and therefore few modules may not be actually available for integration. Integration testing can be performed using two approaches: Top Down Integration Testing or Bottom Up Integration Testing. Stub is a piece of code used during Top Down Integration Testing which simulates the behaviour of the "called program" which is either not available or resource extensive. Figure 3.8 and 3.9 throw light on the concept of stubs used during Top Down Integration Testing of six modules (symbolically named as A, B, C, D, E and F). Similarly drivers imitate the functionality of an unavailable upper level module or the "calling program" and are used in Bottom Up Integration Testing.

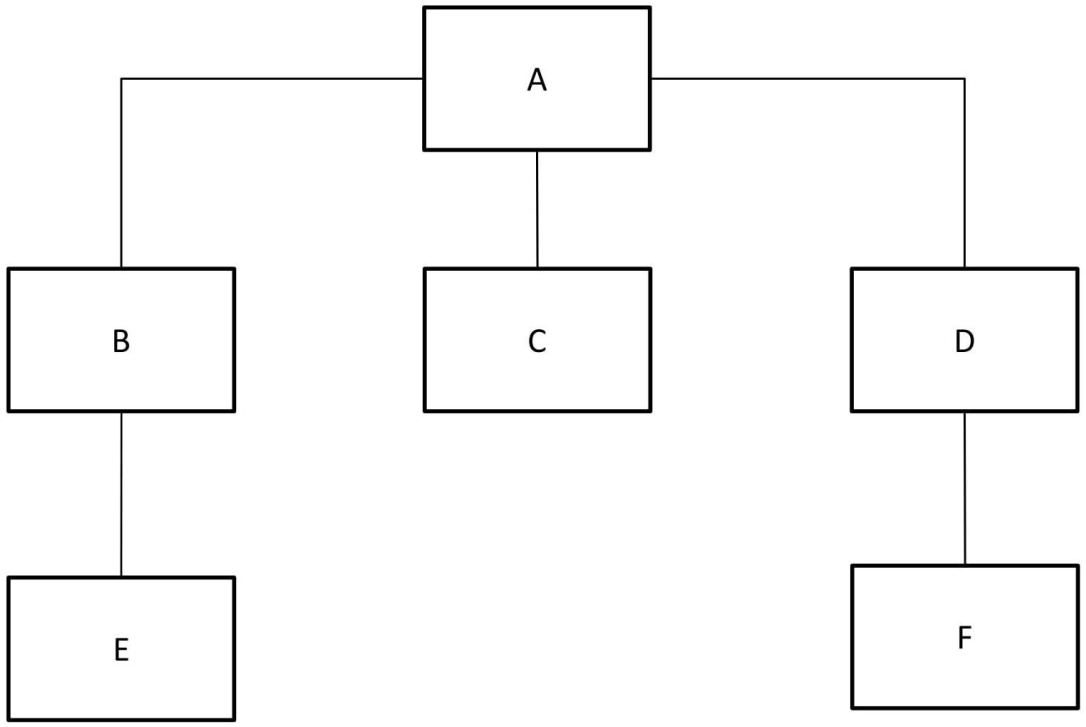

Figure 3.8: Top down integration testing

We propose that aspects can be written to create good stub and drivers and are

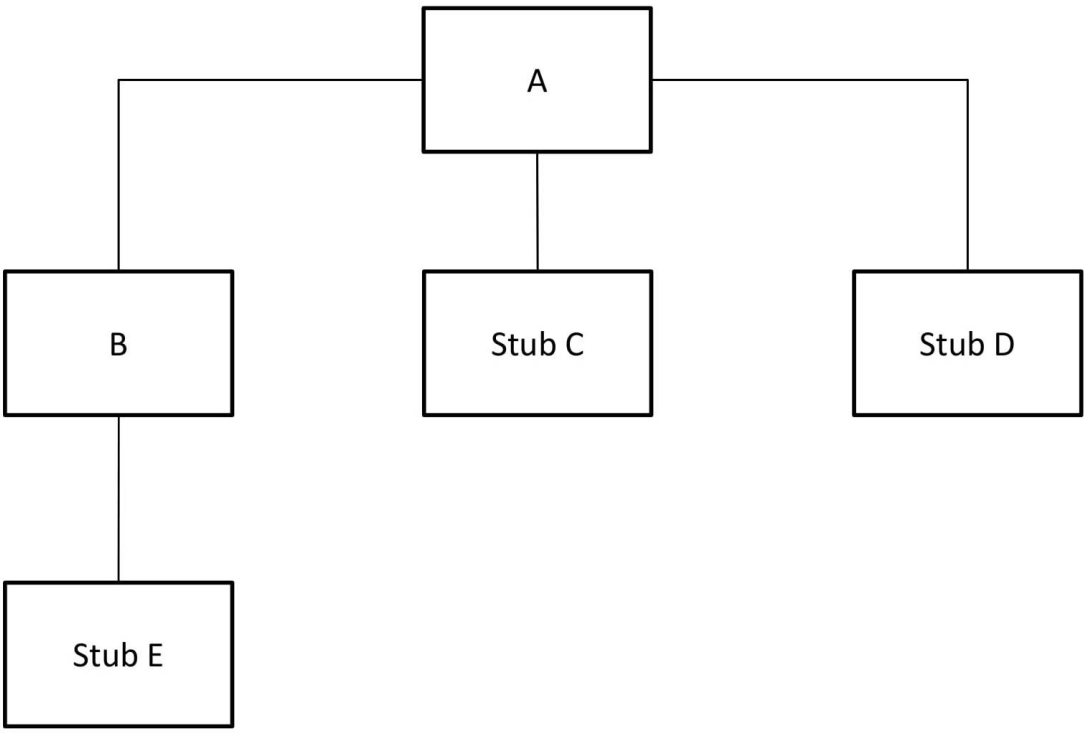

Figure 3.9: Top down integration testing with stubs

thus useful in Integration Testing. We used the *around* advice within aspects to capture the module to be replaced with the stub implementations. For example, consider an application with a Login module which is dependent on the back-end Database module such that the login and password values passed by the Login module are matched in the application database by the Database module and it returns a true on a match (false otherwise). The Login module has been coded and is ready to be tested, but the Database module is not prepared. We created a stub in the form of aspect using the around advice which mocked the functionality of the Database module and returned appropriate value to the Login module [48]. Listing 3.26 shows the source code for our aspect stub.

3.4.3 Acceptance Testing

Acceptance testing can be alpha testing (internal acceptance) and beta testing (external acceptance) which are both possible to be conducted using AOP techniques as described hereunder.

Listing 3.26: Writing stub for Integration Testing using AspectJ

```
public aspect stubAspect {
        String loginArray[]={"tom","john","peter"};
        String passwordArray[]={"12345","abcd","@pqr123"};

        pointcut stub(): call(* Database.checkCredentials(..));
        boolean around(): stub()
        {
                Object args[]=thisJoinPoint.getArgs();
                String login=(String)args[0];
                String password=(String)args[1];
                boolean userExists=false;
                //System.out.println(login + " " + password);
                for(int i=0;i<loginArray.length;i++)
                {
                        if(login.equals(loginArray[i]))
                        {
                                userExists = true;
                                if(password.equals(passwordArray[i]))
                                {
                                        return true;
                                }
                                else
                                {
                                        return false;
                                }
                        }
                }
                return userExists;
        }
}
```

3.4.3.1 Alpha Testing

Alpha testing is basically the term given to internal acceptance testing done at the developer's site which is performed before release of the software and towards the end of the development process. The main motive of alpha testing is to simulate the real users by using black box and white box techniques. The focus is to generate test cases which are similar to the possible inputs by a typical user. Alpha testing ensures that the end users get a high quality application which performs the intended functionalities without errors. The testers are usually the employees of the organisation i.e. in-house skilled engineers who might not be members of the development team.

Most of the testing processes which are carried out in alpha testing can be performed using AOP. As alpha testing is based on either white box or black box testing strategies and since AOP can be used to perform both of them as explained above in Section 3.1 and 3.2, therefore AOP is suitable for performing alpha testing as well. Also as alpha testing is done before the release of the software and when the development work is about to complete, AOP becomes useful to implement the functionality of modules which are yet not fully developed. Although alpha testing is usually performed by highly skilled testers, still if the alpha testers do not have the required knowledge of AOP, our TAGL as explained later in Chapter 5 can be used to generate the testing aspects and execute the alpha test cases.

3.4.3.2 Beta Testing

Beta is the second letter of the Greek alphabet and therefore beta testing is the name given to the external acceptance testing which immediately follows alpha testing. Beta testing is the pre-release testing performed with limited number of external users with the intention to integrate the customer input to improve the quality of the software and ensure release readiness. Beta testing is carried out under real environment and real working conditions with real customers (end users of the software). Testers can be naive or proficient end users of the software. Beta testing typically uses black box testing. AOP is suitable for beta testing because it is carried out using black box strategy for which AOP can be applied as explained above in Section 3.2. Further our TAGL, with its low learning curve as we shall see in Chapter 5, enables the end users to execute complex and extensive beta test cases even though they aren't proficient in the field of software development or testing.

3.5 Agile Testing

Recently the methods for agile software development are getting widely acceptable in the software industry. In agile development, the total software development period is divided into large number of short iterations called *sprints* [56]. The software is then developed incrementally and at the beginning of each sprint, the requirements from customers is analysed and the software (obtained from the previous iteration) is improved to satisfy those requirements. Thus at the end

of every sprint, an incrementally enhanced software is delivered. Requirements analysis in each sprint is the most significant feature of agile software development.

Agile Testing is the practice of testing software for bugs within the backdrop of agile software development process. Since agile processes are based on sprints that do not take into account the future or unknown requirements, agile teams test continuously as this is the only way to be sure that the features expected from a given iteration are properly implemented. Bugs detected in an iteration are fixed within the same iteration and thus the source code remains clean. By coupling testing along with development phase, agile methods produce more robust code more quickly.

Our proposed AOP testing methodology makes agile testing easier. If AOP is used for testing issues which are system wide and effect all or most of the modules of a system, then the testing code shall be confined within the testing aspect only. For example, if a performance testing aspect has been written using wildcard pointcuts which collects the memory usage and the execution times of each function call in the whole program, it will be equally functional without any modifications even when the application is added with new functions in the successive sprints.

Further, incremental changes in the source code at every iteration of the agile process shall require changes to the testing code localised in the testing aspects only. Lets take example of a web service which provides city name upon entering the zip code. In United States, the zip code is that of 5 digits. Now if after one of the sprints, a new requirement is discovered to handle 6-digit pin codes as well (the application is expected to accommodate Indian zip codes too) then we need to perform testing with test cases having 6 digits in addition to the existing test cases. These new test cases can be simply added to the existing array of test cases in the testing aspect (refer Section 3.2). Likewise, crosscutting non-functional properties like security, reliability, performance etc. can be tested incrementally using AOP based testing techniques.

3.6 Smoke Testing

In smoke testing, only the initial test cases are executed in order to check the most important functionalities of the software before going ahead with the complete testing in detail. The name “smoke testing” is derived from the hardware testing phenomenon where it is checked that smoke should not be emitted on the initial

switching of the device. Smoke testing is carried out with test cases that test the major functional areas of the system. The testing team actually builds a set of test cases which should be run every time a new release of the software is developed. The smoke test ideally include at least one test case for every feature or function of the application. Primary features like "creation of a new user", "whether the GUI is responsive or not", "buttons in a window work as intended or not", "successful login", "successful logout" etc. are tested under the umbrella of smoke testing. We executed the aforesaid smoke tests cases using aspects and found that AOP is suitable for smoke testing as well.

3.7 Regression Testing

Regression testing is carried out when some changes are made to the existing application. Regression testing is required in one of the following cases:

- When new features are added or existing features are deleted from the application
- When the software is adapted in accordance with the new environment and conditions
- When a discovered bug is fixed, it might accidentally result into introduction of new bug(s)
- When an optimisation is carried out in the application

Whenever such a change is made in an application's module, the test suite associated with the module has to be modified: either new test cases are to be added replacing the existing test cases and/or the existing test cases need to be modified. It is so because when a software S is modified to S', all the tests developed for S may not be always applicable for S'. Foremost reason is that there is not enough time available to run all the tests. Further, the input data or its format for S might be different from that of S'. Moreover, the output expected from S'could be different from what was originally expected from S.

With our proposed AOP based testing, the tester can select and modify the test code written within aspects according to the change in the application [57]. New

test cases (in addition to the older test cases) can be added to the testing aspect. Moreover, the tester can test whether state invariant is preserved or not after bringing up the new changes by using the *set* and *get* pointcut. For example, if after bringing about a modification in an application's module, the tester suspects that a particular variable in the module might have been affected, then the *set* pointcut can be used to verify that the values of the variable still remain within the desired constraints whenever its value is written in the module as shown in Listing 3.27. Similarly the *cflow* pointcut can be used to test for any incorrect modification in the control dependencies. In AspectJ, we can use *cflow* to determine the joinpoints that fall under the control flow of a particular pointcut. We used this to ascertain that the same control flow is maintained even after making the changes in the application.

Listing 3.27: Regression testing example

```
public aspect regressionTest {
        int min=_expectedminvalue_, max=_expectedmaxvalue_;
        void around(int datamember_value) : set(int
            ↪ classUnderTest.datamember) && args(datamember_value)
        {
                if (!( datamember_value>=min &&
                    ↪ datamember_value<=max))
                {
                        System.out.println("Test failed");
                }
        }
}
```

3.8 Summary

In this chapter, we discussed about the different types of software testing and proposed the use of AOP technique for carrying out these. Suitable illustrations of Java programs and their testing using aspects in AspectJ have been provided in the chapter to substantiate our proposed testing methodology. It can be inferred from the discussion in this chapter that AOP is suitable to carry out most of the types of software testing and that too in a non-invasive manner.

Chapter 4

Applying AOP Approach for Testing Open Source Applications

In order to establish the benefits of using AOP for testing and to evaluate our approach over conventional testing techniques, we carried out different types of testing of few widely used open source software. As AspectJ is the de-facto standard for AOP created for Java programming language, we selected projects like NetC, JDownloader, JScreenRecorder, JFreeChart, JGAP etc. which are all written in Java and are available in open source community. The total number of downloads of all these software has been quite significant which signifies their popularity. Using our proposed AOP methodology of testing, we could detect remarkable bugs into these software.

The main idea behind applying our AOP approach on testing of open source applications was to analyse the effectiveness of our approach. We performed various types of testing on the aforesaid open source applications rigorously like black box, memory leakage, load testing etc. with numerous input test cases. We intended to find out bugs using our proposed AOP technique of testing and establish the usefulness of this approach. As an outcome of this work, we could find out phenomenal bugs into the aforesaid applications which we notified to their developers through their bug reporting forums or via email. All the bugs reported by us were acknowledged and almost all were considered as important bugs by the developers of their respective applications. The developers not only acknowledged but also assured to rectify the reported bugs in their future releases.

In the following sections, we provide brief description about the open source applications that we used for the purpose of evaluating our approach. The date when

last updated, lines of code, number of downloads etc. of these applications indicate that their uses are quite widespread. Further, we discuss the procedure how we performed testing using our AOP technique through which we could discover bugs in these applications. We provide details about the module in which the bugs were discovered along with their causes. Also the comments of the developers on our bug reports have been spelled out.

4.1 Testing NetC

NetC is a popular chat software that can be used over local area network (LAN) and was last updated on 23rd April 2014. It is multi-platform and works on different operating systems. It provides all the necessary and basic chat functionalities like group chat, smiley, user status, file transfer etc. NetC has been claimed as a server-less chat software by its developers, i.e., no server is required for its operations [58]. The total number of lines of code in NetC are 5510.

NetC is available open source and has got 33 different Java classes. Till date (30th January 2018), it has been downloaded 43641 number of times since its upload on *www.sourceforge.net*. The number of downloads in the last year (2017) have been 771. The number of download of this application has been significant so far which prompted us to choose this software for applying our proposed approach and perform different types of testing of its different classes. We used AspectJ to write the testing aspects.

While performing the load testing of the *ChatConnection* class of NetC using the AOP technique on a system with Windows XP Service Pack 3 (SP3) having Intel T6670 Processor and 4GB RAM, we observed that the application is not able to handle excessive user load in a professional way. The load testing aspect written for this purpose is shown in Listing 4.1. The aspect with an *after* advice which executed after the *connect* method of the *ChatConnection* class, creates multiple users inside a for loop. For every new user created, a new socket is opened. During such load test, we observed that until we keep the number of newly created users limited to 1500, the software works fine. But after that as we increase the number of users and reach to 1800 users, the software starts hanging and the same behaviour continues up to 4000 chat users as shown in Figure 4.1. Beyond 4000 users, the software is not able to create more users and starts giving the *Connection Refused* exception unexpectedly. In fact, such a case of excessive

load should have been handled rightly with a proper message to the new users which attempt to use the application after the possible limit.

Listing 4.1: AspectJ: NetC Load Testing

```
public aspect loadTestingAspect {
        pointcut chatServerConnection() : execution(public void
            ↪ com.dgtalize.netc.net.ChatConnection.connect());
        after() : chatServerConnection ()
        {
                int port=41517;
                int i=0;
                //Load can be varied by changing value of n
                int n=100;
                for(i=0; i < n;i++)
                {
                        try
                        {
                         Socket socket = new Socket("localhost", port);
                         ByteBuffer paquete = ByteBuffer.allocate(512);
                        DataOutputStream sockOut = new
                            ↪ DataOutputStream(socket.getOutputStream());
                         sockOut.write(paquete.array(), 0, paquete.limit());
                        }
                        catch (UnknownHostException e)
                        {
                         e.printStackTrace();
                        }
                }
        }
}
```

Secondly, we also carried out black box testing of the *PrivateMsgWindowUI* class of NetC which provides functionality of private messaging to a specific user available (see Figure 4.2). When a user provides the name of another user to send the private message to, then the *validateInput* method of the *PrivateMsgWindowUI* class validates whether such user exists or not. If the user exists, a new private chat window is opened or else if the specified user doesn't exist, an error message is shown to the user. We tested the *validateInput* method with various inputs as shown in the pointcut in Listing 4.2 and found that the method produces error message when the user doesn't exist and the input is of alphabetical nature. Even the special character inputs (including null and blank spaces) have been handled well but the applied validation fails to give any error message at all when the input is numerical like "123" etc.

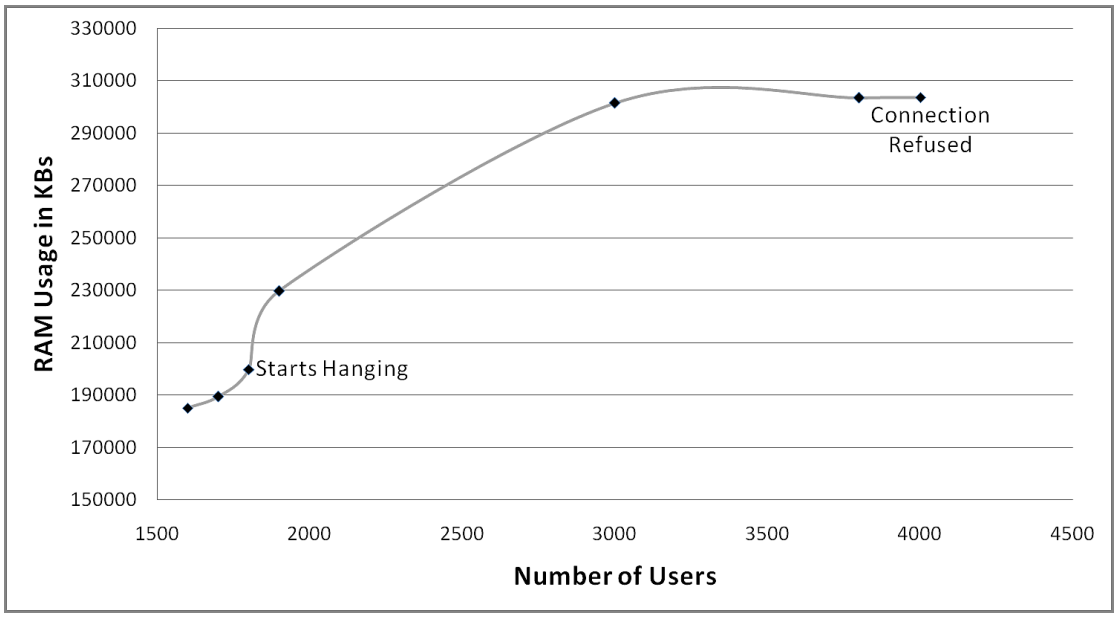

Figure 4.1: Load testing NetC by creating multiple chat users using aspect, performed on a system with Windows XP SP3 having Intel T6670 Processor and 4GB RAM

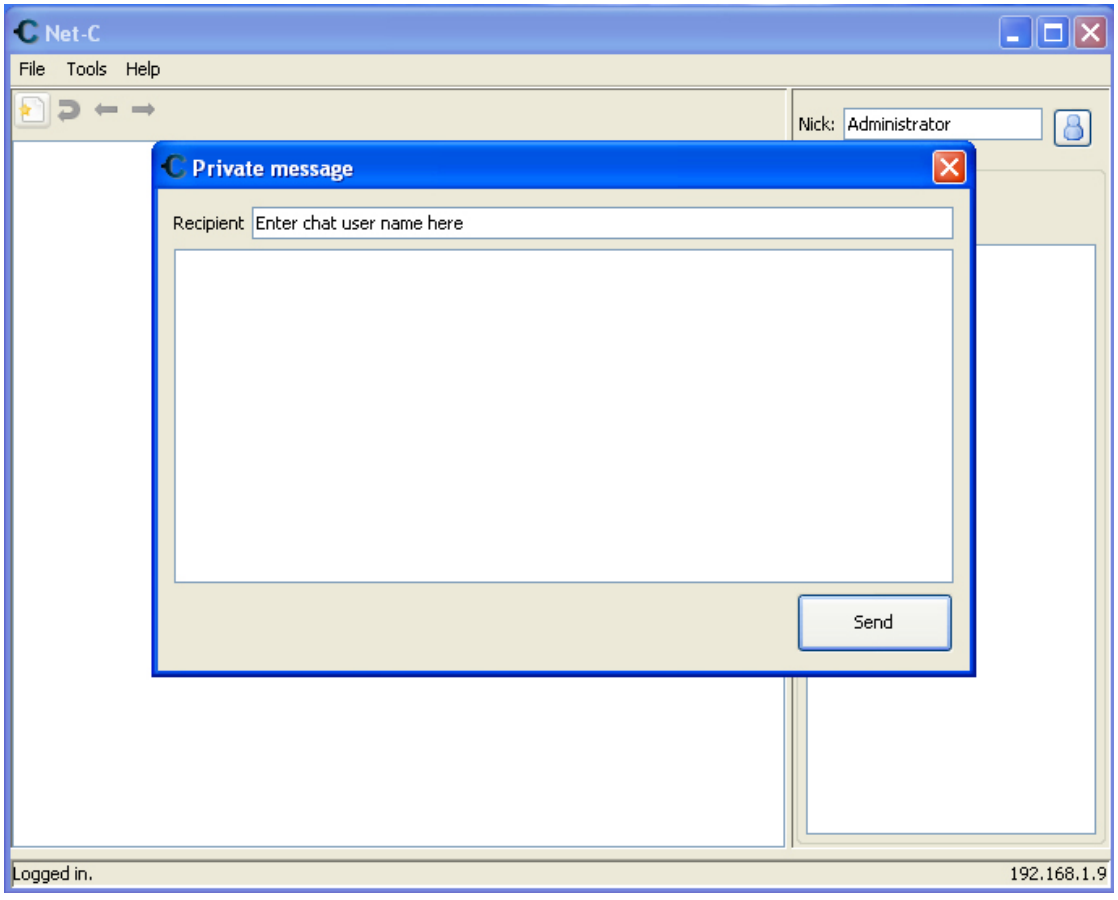

Figure 4.2: NetC: Private Message Window

Listing 4.2: AspectJ: Testing Input Validation

```
pointcut
    ↪ userNameValidationTest(com.dgtalize.netc.visual.PrivateMsgWindowUI
    ↪ p) : execution(private boolean
    ↪ com.dgtalize.netc.visual.PrivateMsgWindowUI.validateInput()) &&
    ↪ target(p);
        boolean around(com.dgtalize.netc.visual.PrivateMsgWindowUI p) :
            ↪ userNameValidationTest(p)
{

        String saveInput = p.recipientText.getText();

        String[] testinput = new String[] { "Ram", "abcdefghijklmnop", ".",
            ↪ "***", " ", null, "123" };
        for (int i = 0; i < input.length; i++)
        {
                p.recipientText.setText(testinput[i]);
                boolean result = proceed(p); // invoke test
        }
        p.recipientText.setText(saveInput);
        return proceed(p); // do original processing
}
```

We reported both of these issues to the developer of NetC (namely, Diego Gurpegui) at email to which he responded by writing: *..it seems that the errors you reported could in fact happen.*

4.2 Testing JDownloader

JDownloader [59] is an open source download management tool written in Java which simplifies the process of downloading files for Internet users. JDownloader provides with provisions for starting/stopping/pausing downloads, theme selection, setting up the bandwidth limitations, auto-extraction of archives and filtering the downloaded contents. The two main advantages of using JDownloader is that it boosts the speed of the Internet downloads and that it reduces the time it takes to start a download. Also the features and interface of this application are easily comprehensible.

JDownloader has got 973165 lines of source code and is maintained by a large development team. From *www.sourceforge.net* alone, till date (30th January 2018)

it has been downloaded 30858 times and there have been 476 downloads in the last year (2017). In 2009, the website of JDownloader http://www.jdownloader.org/ was the 100 most visited website of Spain and it was ranked among the top 50 downloaded applications. There are different versions available for download for Windows, Linux and Mac operating system. For all these reasons, we considered JDownloader as a good candidate for analysing the application of our approach.

We could perform various types of testing of the different classes of JDownloader using our approach and found one important bug in the *AddLinksDialog* class. The *AddLinksDialog* class of JDownloader provides us with the functionality of adding a new link and downloading all the downloadable items from this link. The GUI snapshot of this feature is shown in Figure 4.3. We performed the black box testing of this class using our aspect approach. For this, the *validateForm* method of the *AddLinksDialog* class was instrumented using an around advice. As the *validateForm* method is protected, we used *privileged aspect* to capture it. The *input* member of this class was set to various Uniform Resource Locators (URLs) to test the proper working of this functionality.

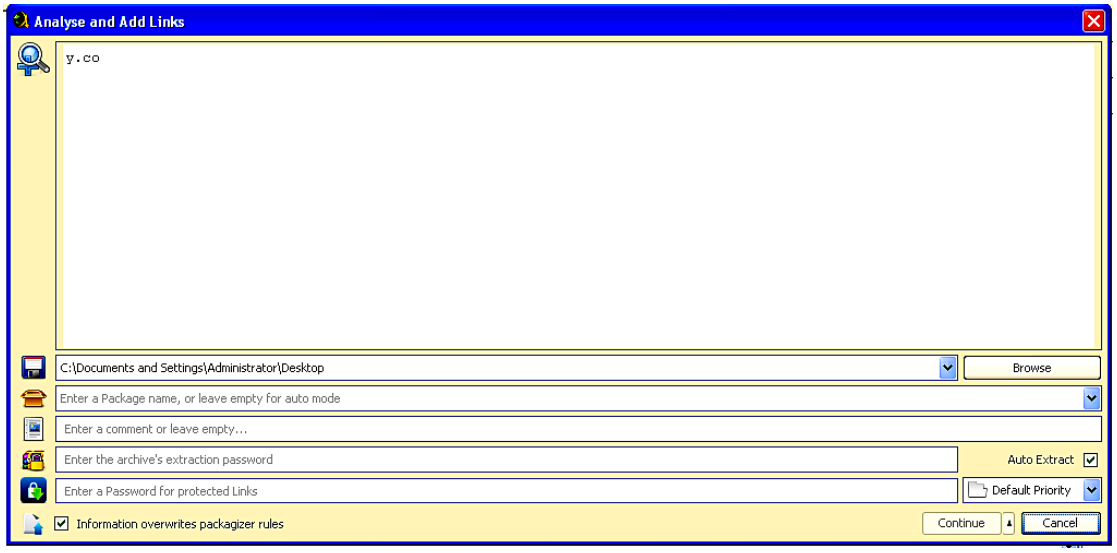

Figure 4.3: JDownloader: Snapshot of *Analyse and Add Links* GUI

JDownloader worked fine with almost all URLs except the small single letter URLs to which the *Continue* button for starting the download did not get enabled (see Figure 4.3). When input URLs like "t.co" (twitter's shortened link), "y.co" (popular yacht selling website), "c.tv" (Internet future technology innovators), "i.tv" (TV guide app for iPhones) were used, the *Continue* button did not get enabled and thus the JDownloader function of downloading from the provided link could not be used. We could infer that this functionality won't work for single letter

domain names at all and there are numerous examples of such single letter premium domains. We reported this bug in the JDownloader's bug reporting forum to which their developers responded, accepted and opened a new bug # 82418 in their bug tracker system (see Figure 4.4). Further, JDownloader exhibits memory leakage problems (excessive RAM usage) after downloading large number of files which we observed by load testing it with numerous download links that we input using aspects.

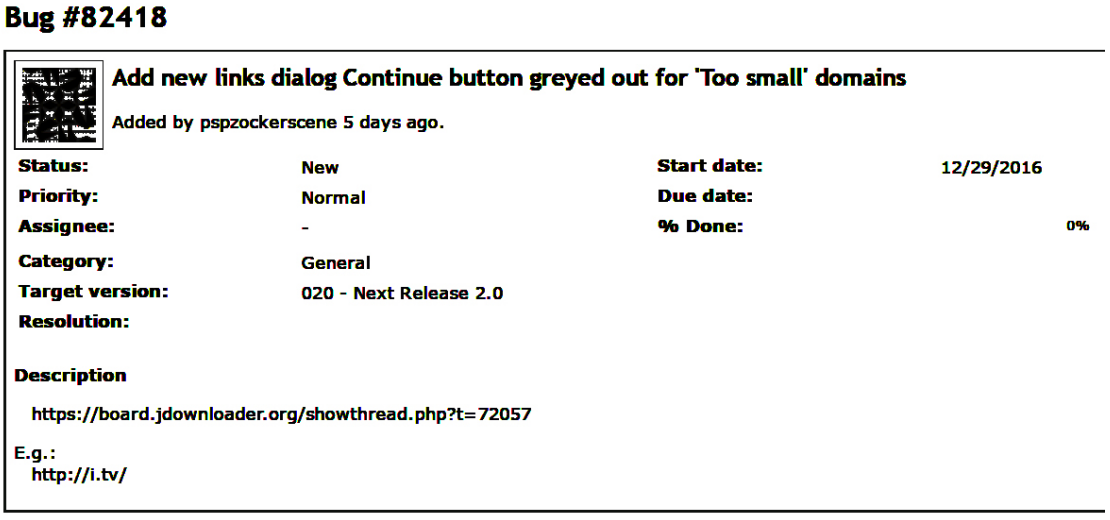

Figure 4.4: JDownloader: Snapshot of bug acceptance

4.3 Testing JScreenRecorder

JScreenRecorder [60] is an open source Java application available under the Lesser GNU Public License. It provides the functionality of screen recording using which we can record specific part of or the complete Windows desktop screen in the form of a video file. We can choose from a list of 128 mouse cursor designs. Custom textual watermarks can also be created. Mouse cursor and watermark both can be included in the recorded video. The application runs in a compact window and the features can be availed through its self-explanatory icons. Before starting the recording of the screen, few settings like frames per second, path of output directory etc. can be set. The recorded video is encoded in MP4 format.

The software is available at popular open source projects' sites like sourceforge, github etc. We downloaded the source files of JScreenRecorder from *www.sourceforge.net*. The application has got 30 Classes and 3863 lines of Java code. Till date (30th January 2018), it has been downloaded total 22,422 times

from www.sourceforge.net alone, with its maximum downloads from United States (29% of total). It was last updated on 20th November 2015.

We performed various types of testing of JScreenRecorder using our AOP approach. While performing the simulation of mouse pressed event using our testing aspect, we detected a remarkable bug into the application which we also later intimated to its developer Deepak P K. JScreenRecorder provides with a "select capture area" button using which the part of desktop screen that has to be recorded can be selected. In the JScreenRecorder application, this functionality has been implemented in the *RecordControlPanel* class. When we tested the *RecordControlPanel* class with our testing aspect by instrumenting the *captureAreaButton_MousePressedEvent* method with an around advice as shown in Listing 4.3, we observed that even after one "Capture Area Selector" form is open, still another such form can be opened by passing a *MouseEvent* (see Figure 4.5). Although this should not be the case. Since the JScreenRecorder application allows only one screen recording operation at a time, opening two such forms is of no use. Moreover, only the last opened form is considered by the software for deciding the screen space for recording. The expected behavior is that once a "Capture Area Selector" form is opened, the "select capture area" button should either be disabled or else an error message should be generated when another *MouseEvent* is passed.

Listing 4.3: JScreenRecorder mouse pressed event simulation

```
pointcut pcMousePressEvent(java.awt.event.MouseEvent evt) :
    ↪ execution(private void
    ↪ captureAreaButton_MousePressedEvent(java.awt.event.MouseEvent))
    ↪ && args(evt);
void around(java.awt.event.MouseEvent evt) : pcMousePressEvent(evt)
{
        //Create a new mouse event to simulate a mouse press
        java.awt.event.MouseEvent e = null;
        //Pass the new mouse event as argument
        proceed(e);
        //Proceed with the original mouse press event to see whether the
            ↪ application generates an error message
        proceed(evt);
}
```

Similarly, while testing for null pointer exception handling in JScreenRecorder, we observed that null values have not been well dealt with in the various classes

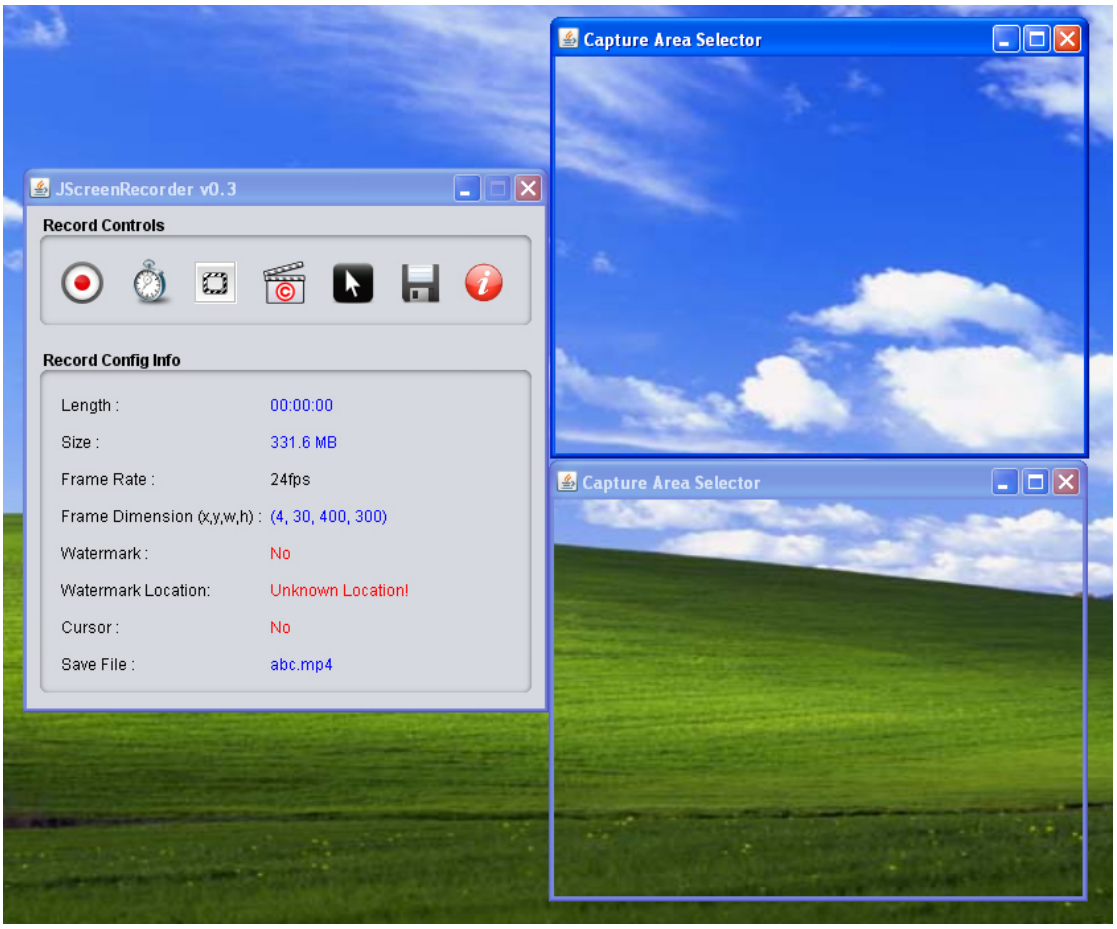

Figure 4.5: JScreenRecorder: Snapshot of multiple *capture area selector form* opened simultaneously

of JScreenRecorder. For example, we tested the *setVideoLength* method of the *RecordConfig* class to verify how shall it handle if, for some reason, the video length happens to be null. There could be cases like when disk space is full or when there are insufficient permissions for the destination directory, the video shall not be recorded and its value could be null. The testing aspect shown in Listing 4.4 passes a null value as argument to the *setVideoLength* method upon which the application raises a *java.lang.NullPointerException* as there exists no mechanism for handling the null values.

We posted both of these observed bugs (multiple capture area screen issue and unhandled null pointers) through email to the developer of JScreenRecorder on 25th December 2016 and the bugs were acknowledged. Developer Deepak P K responded to our email and wrote: *Thanks for your feedback. Will look into it and make appropriate changes for the next release.*

Listing 4.4: JScreenRecorder null pointer handling test

```
public aspect testNullHandling
{
  pointcut testNullPC(long videoLength): execution(*
      ↪ com.jscreenrecorder.core.config.RecordConfig.setVideoLength(long))
      ↪ && args(videoLength);
  void around(long videoLength) : testNullPC(videoLength)
  {
                videoLength = (Long)null;
                proceed(videoLength);
  }
}
```

4.4 Testing JFreeChart

JFreeChart [61] is an open source Java library that supports a wide variety of charts which can be used by the developers in their applications. JFreeChart supports pie charts (2D and 3D), bar charts (horizontal and vertical, regular and stacked), line charts, scatter plots, time series charts, high-low-open-close charts, candlestick plots, Gantt charts, combined plots, thermometers, dials and more. JFreeChart can be used in client-side and server-side applications. This project is maintained by David Gilbert [62].

The latest available version of JFreeChart is 1.0.19. It can be easily downloaded from popular project hosting repositories like github, sourceforge, mvnrepository etc. Since its upload in 2001, it has been downloaded 4,303,727 number of times from sourceforge alone which clearly indicates the usefulness of the library. This open source application has got 319,071 lines of Java code. The total number of source files are 1017.

We performed various types of testing of JFreeChart using our AOP approach. During the memory leakage testing, we detected a memory leak in the *ChartPanel* class. Various objects of *Graphics2D* are created within this class for which it is necessary to call the *dispose* method which abandons this graphics context and releases the system resources used by it. When we used our "testMemoryLeak" aspect (as illustrated in Section 3.1.1 of Chapter 3) to test the creation and disposal of all Graphics2D objects in this class, we observed that although a total of 5 objects of Graphics2D class are created for which the dispose method should be called, it has been called only 4 times. For one of the Graphics2D objects, namely *bufferG2*, created within the *paintComponent* method of the *ChartPanel* class,

the dispose function has not been called which becomes reason for a memory leak. Moreover, this leak shall lead to aging issues with the application's execution over time. The source code for the *ChartPanel* class is provided at Appendix A.

We posted this memory leakage bug at github on 6th February 2017: *...we observed that although a total of 5 objects of Graphics2D class have been created in this class for which the dispose method should be called, it has been called only 4 times....* David Gilbert replied: *Thanks for the report Manish. I fixed it, details you can see via your bug report at GitHub: https://github.com/jfree/jfreechart/issues/38.*

Another small but important issue that we observed while black box testing of the *createPieChart* method of the *ChartFactory* class in the JFreeChart application. When we provided negative value in the dataset to be used for the testing of *createPieChart* method, it was simply ignored while creating the pie chart and the pie chart was plotted simply using the rest of the positive values of the testing dataset. Contrary to this behavior, the application should have generated an error message as it is impractical to plot a pie chart with negative values.

We posted this bug as well at *github* on 7th February 2017: *In the createPieChart method of the ChartFactory class: if a negative value is provided in the dataset to be used for the pie chart, it is simply ignored while creating the pie chart and JFreeChart plots the pie chart using the rest positive values of the data set. Contrary to this behavior, the application should generate an error message as it is impractical to plot a pie chart with negative values.* To this David Gilbert responded by writing: *I think you are correct that it is better to "fail-fast" if the pie dataset contains negative values.*

4.5 Summary

In this chapter, we have discussed regarding the practical utilisation of the proposed testing technique using AOP on open source software projects whose usages are widespread. Different types of testing were conducted using aspects and significant bugs were detected in the selected software projects. The positive acknowledgements from the developers and the bug report forum managers of these selected software projects established the usefulness of our proposed testing technique for conducting various types of software testing.

Chapter 5

Testing Aspect Generator Language

Domain specific languages (DSLs) are little languages tailored for a particular purpose which shield the users from much of the complexity of explicitly programming in the general purpose languages (GPLs). A GPL is suited to multiple domains whereas a DSL is designed for a particular domain. There are various DSLs which are widely used in specific domains like Structured Query Language (SQL) for query writing, Latex for document writing, Verilog for hardware description and many more. Important application domains for which DSLs have been created by researchers and developers are listed in Table 5.1. Writing DSL for a particular domain is worthy because programming using a well designed DSL is much easier and it also helps collaboration between the programmers and the experienced domain experts. Our proposed *Testing Aspect Generator Language (TAGL)* is presented in this Chapter. TAGL is a domain specific language that is specifically useful in the domain of testing of Java applications by automatically generating the testing aspects. TAGL has been implemented using *lex* and *yacc* and can be used for the purpose of conducting various types of software testing.

Table 5.1: DSL Domains

Pattern Matching	Job Scheduling
DBMS	Style Sheet
Hardware Description	Parsing
Text Processing	Telecommunications
Graphs	UML Diagrams
Web Layout	Visual Modeling
Document Layout	Logic
Robotics	Graphics

5.1 Why Domain Specific Language?

A DSL is designed in such a way that the available language constructs are easy to remember. DSL thus enhances the speed of development and also the ease of use. For example for most DSLs, natural language is used as syntax. Moreover, syntax is chosen such that it best fits the problem. DSLs are created for easier human understanding and therefore they are made such so that human can conveniently edit and carry out the necessary development out of real words. In a nutshell, DSLs are designed to render a natural way to express the solutions to specific problem spaces and thus improve the productivity of the software developers.

In order to understand how DSLs make life easier for the developers, let us take example of *regular expression* which is a commonly used DSL [63]. The following regular expression is suitable to match most email address patterns:

$$[A-Z0-9.\_\%+-]+@[A-Z0-9.-]+.[A-Z]\{2,4\} \tag{5.1}$$

This regular expression does not appear to be very cryptic. Imagine if the same logic is coded using a GPL, it shall require several lines of code. And further it shall also be cumbersome to understand and modify that code if required at a later stage. On the contrary, the above regular expression is quicker to write, easier to comprehend, simpler to modify and less likely to be bug-prone. From this example, it is apparent that a well written DSL can save a lot of developer's efforts.

As most of the times, DSLs are written in form of natural language, the programs written therein become easily comprehensible for the end customers as well, who are generally non programmers. Thus by the way of DSLs, the application code can be exposed to domain experts who may not have programming knowledge but understand the business logic very well. If a domain expert can read and understand the application code then he or she can aptly guide the coder regarding the expectations, correctly understanding which had always been a great difficulty for the coders. However, such goals using DSLs are achievable only by a proper and elegant design.

Thus to summarise, DSLs are particularly important for the following reasons:

- Make development easy

- Enhance the speed of development
- Less error-prone code
- Simplify the maintenance of code
- Helps improve the communication between the coders and the end customers

The above listed benefits resemble the benefits as claimed for *high level languages*. Thus, it can be inferred that DSLs are simply *very high level languages* [64]. Further we would like to emphasise that the overall cost incurred in the process of software development is less when we use DSL as compared to GPL as depicted in Figure 5.1. In the initial phases of the software development life cycle, when the DSL is designed and implemented for the particular application domain, the cost is high for obvious reasons. But as we move ahead in the development cycle, the cost for software maintenance and testing reduces considerably. This is so because code written using DSL (which generally resembles natural language) is easier to read and understand. Also as the DSL code is easily comprehensible, it increases the collaboration between the developers and end customers. This further reduces the development cost as the customer requirements can be understood well. Thus aggregate cost for software development using DSL is considerably less.

5.2 Types of Domain Specific Languages

DSLs are mainly of two types: external and internal [65]. External DSLs are those which are parsed independently of the host GPL. Cascading Style Sheets (CSS), Cucumber, Regular Expressions are examples of external Domain Specific Languages. They are not built on top of any language, rather they have a syntax of their own. A good parser is required to be built for such external DSLs which understands the language and translates it to another one. External DSLs are quite common in the Unix/Linux community. On the contrary, internal DSLs are built on top of the host GPL. In other words, DSLs that modify the host language in such a way that makes them different and useful for a specific domain are internal DSLs. The syntax of the internal DSLs is mostly restricted by the host language. Development in internal DSL's is done through a particular form

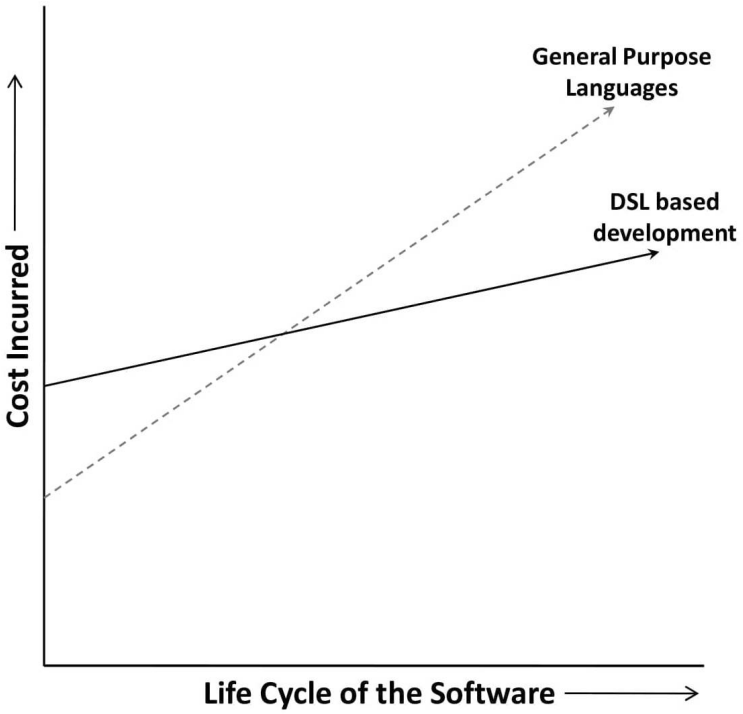

Figure 5.1: Cost incurred using DSL vs. GPL
[64]

of API in the host GPL. Unlike external DSLs, the user is not required to learn the grammar for internal DSLs. JMock is a good example of internal DSL.

There are various ways a DSL can be designed and implemented [17]. Tools like Scala, F# etc. can be used to write internal DSLs. External DSLs can be written with tools like Eclipse Xtext, Lex-Yacc etc. When using lex and yacc, one develops a lexer and a parser which then generate the executable code in some GPL. Taking use of lex and yacc makes the job of writing a DSL easier as compared to writing the same from scratch. Further, use of lex and yacc gets the job done quickly and in a more maintainable way. We have used lex and yacc in our work to implement our external DSL named *Testing Aspect Generator Language* for the automatic creation of the testing aspects.

5.3 Learning curve of testing tools and TAGL

Learning curves have been studied for decades in order to reduce the cost factors. Figure 5.2 shows the comparison of the learning curve of a DSL with that of a GPL. It is evident from the figure that the length of time required to acquire a proficient skill set or a high level of comprehension of DSL is considerably less.

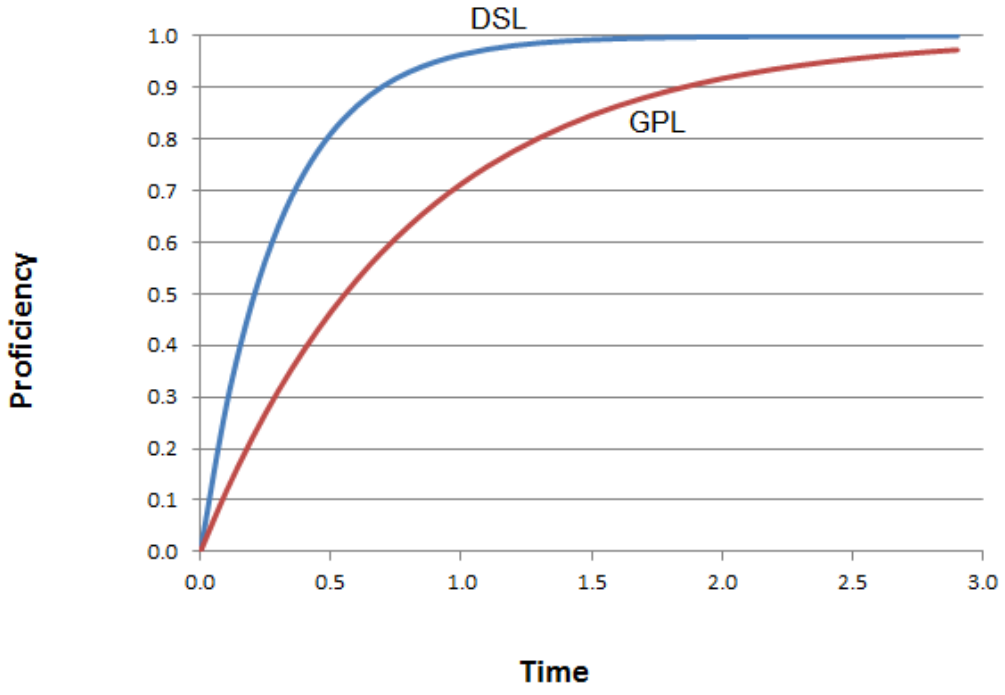

Figure 5.2: Learning curve of DSL vs. GPL
[66]

Learning curves not only apply to software development or software usage but also to software testing. Technical skills are required to accurately design and further maintain the test automation framework and the test scripts. As noted by Rafi et. al [28], one of the important disadvantages of automated testing is that the testers need to acquire the required level of proficiency for writing the testing code using the automation tool. Besides the need for domain knowledge, the "tools and techniques" and the "process and methods" also need to be learnt [67]. There are learning curves associated with all testing tools; for few like JUnit the learning curve is medium, but for others likes Selenium the learning curve is high, i.e., requires more learning efforts [68]. In fact, the AOP approach, proposed by us in Chapter 3 for conducting various types of software testing, requires the tester to acquire the programming skills necessary for writing the testing code using AOP. Since AOP is a new programming paradigm, not all developers or testers may be familiar with this technology and therefore using it practically shall entail certain learning curve.

Such learning curves for software testing effect the complete software development cycle and software time-lines and budgets can't be matched. Usually the time and budget available for testing are constrained and thus if a domain specific language

is designed that reduces the efforts required for learning the testing tools and approach, the actual testing process of bug finding can be accelerated. In this chapter, we describe one such DSL devised by us which we have named as Testing Aspect Generator Language (TAGL). TAGL has been implemented keeping in mind the specific domain of testing Java applications and has got quite an intuitive and natural language-like syntax. Thus it reduces the learning curve associated with testing Java applications. Using TAGL, even the Java testers who do not have expertise in AspectJ can still avail the benefits of testing Java applications using AspectJ.

As we shall discuss in the next Section 5.4, TAGL statements are written in the form of comments and thus do not effect the compilation behaviour of the original source code. These comment-like statements are parsed by the lexical analyser and parser which have been written using lex and yacc as we shall discuss in Section 5.5. This in turn produces the testing aspects in AspectJ language which are used to test the system under test. Our TAGL is specific to our proposed approach and easy to learn and write. Learning a new automation tool is indeed a time taking process. This can be avoided by selecting a testing tool which offers minimal learning curve and our TAGL is one such tool, which shall be apparent from the discussion regarding its straightforwardness and utility in the following sections.

5.4 TAGL Syntax

Using our TAGL, the testers who do not have the knowledge of AspectJ can write the testing code for testing Java applications in the form of TAGL statements. Moreover, the syntax of TAGL is quite instinctual, natural language-like and based on fixed patterns.

In TAGL, all the statements are to be written starting with "////". As our TAGL is only meant for generating testing aspects, it is desirable that the TAGL statements does not cause any changes to the source program under test. Since statements written with "//" in Java are considered as comments and since "////" is used as start pattern for the TAGL statements, therefore such TAGL statements shall not be compiled and thus shall not effect the compilation behaviour of the source code. Further, we use these *forwarded slashes-duo* twice in order to distinguish these from the comments written by developers. The lexical analyser and parser have been implemented in such way that every statement that starts with "////"

is interpreted as a TAGL statement and other statements are simply ignored and no action is taken.

All the statements in a particular TAGL *denotation*, that is meant to generate one testing aspect, are to be written in running fashion, i.e., one after the other. Further, the statements in a TAGL denotation are of the form ////itemname: itemdescription. Here "itemname" signifies the entity that has to be described whereas "itemdescription" contains the values or description of this entity. The colon is used to separate the two. For example, in order to specify the signature of a simple *display* method that has to be tested, the corresponding statement in TAGL denotation would be:

$$////methodsignature : public\, void\, display() \tag{5.2}$$

Similarly, TAGL statement to particularise the expected outcomes (for different inputs) from a method can be written like this:

$$////expected : 0, 1, 10, 1000, -1, 4096 \tag{5.3}$$

The first line of a TAGL denotation signifies what type of testing is to be performed by the corresponding testing aspect. The item name to denote the type of testing is *type* and then after we provide the name of type of testing in the description which is written after the colon (:). For example, if the TAGL denotation is to be written for carrying out fuzz testing, then the first TAGL statement would be of the form:

$$////type : fuzztesting \tag{5.4}$$

Likewise, there are different TAGL statements which are written at the start of a TAGL denotation to indicate the type of testing to be performed. For performing different types of testing, different value of *type* has to be used. The following TAGL statement, i.e., the second statement is used to provide a name for the testing aspect that shall be generated. To provide a name for the generated testing aspect, the TAGL statement shall be of the form:

$$////aspectname : fuzzingAspect \tag{5.5}$$

However, such statement for naming the generated testing aspect is optional. If the statement exists, then the generated aspect is given the name specified in

the statement or else a default name based on the type of testing provided in first TAGL statement is given, which is of the form type_of_testingAspectN. Here the suffix N denotes the number of generated testing aspect. For example, fuzztestingAspect1. If it is the first aspect of a particular testing type, then the suffix shall be 1 and for further aspects meant for similar type of testing, this value shall continue to increase.

Further, the rest of TAGL statements are written to manifest the complete testing aspect. The syntax to be used for writing the TAGL denotation for different types of testing is simple and quite easy to learn as we shall demonstrate in the forthcoming subsections.

5.4.1 TAGL for Creating Black Box Testing Aspects

The primary objective of carrying out black box testing is to assess that the application or its component under test does what it is supposed to do. In black box testing, the tester is usually not the programmer or author of the program. The tester has access to the artifact's external interface only and the internal state cannot be examined. The results of the tests are observed in the artifact's output. Thus, in black box testing, the tester can be even non-technical as he/she is only expected to focus on the user actions or inputs [69].

Black box testing is performed to see if the application (or its component) under test meets the user's requirements and thus, specified inputs are provided and the actual output obtained is compared with the expected output as per the software requirement specification (SRS). The inputs provided can be valid or even invalid and tests are conducted for various combinations of such inputs. The intent of a black box tester is to devise all possible set of input conditions that shall fully exercise the functional requirements.

As we explained in Chapter 3, AOP can be used to perform black box testing. In the AspectJ example provided in Listing 3.12 of Chapter 3, the around advice in the testing aspect replaces the execution of the function that matches the defined pointcut. Further arrays of input combinations are created and then *proceed* is called with these inputs one by one to test the desired function. Also the output obtained can be captured and compared with the expected output as per the SRS.

As the black box testing is mainly based on providing inputs to the external interface of the program under test, we have devised our TAGL such that it can be

used by the tester to specify the testing inputs. The TAGL denotation specified by the tester is automatically converted into the testing aspect (by the lexical analyser and parser) with the required pointcut and around advice. As an example, for performing black box testing of a simple two argument function "ComputeInterest" of the "Banking" class written in Java which calculates interest for a year, we can write the TAGL in the following way:

////type: blackbox
////aspectname: blackBoxTestComputeInterest
////classname: Banking
////methodsignature: public float ComputeInterest(float principal,float rate)
////argumentname: rate
////values: 0, 1, 7.5, 10, 1000, -1, 4096
////expected: -1, 1000, 7500, 10000, 1000000, 4096000

In the above TAGL denotation, the *////type: blackbox* indicates that the testing to be performed is black box testing. The following statement is optional in which the tester may specify the name to be used for the testing aspect that shall be generated. In this TAGL example, we have provided *blackBoxTestComputeInterest* as the name to be used for the generated testing aspect. If no name is specified, the automatic aspect generator generates a default name for the testing aspect which shall be blackboxAspect1 in this case. Next statement specifies the name of the class which is to be tested. The fully qualified class name (along with package structure, if any) has to be provided. Then after the tester specifies the signature of the method that has to be black box tested. Using *////argumentname:*, the tester provides the name of the argument for which the method shall be executed for different input values and tested for proper functionality as per the requirement specification. Next the tester provides with the various values that are to be used for the specified argument (each one by one) to test the specified method. Optionally, if the tester is willing to compare the actual output with the expected output as per the requirement specification, which is mostly the case, then the expected outcome for each input value can be provided using the keyword *////expected:* . If such expected outcome values are provided by the tester, then our lexical analyser and parser generates a function in the output testing aspect that compares every actual outcome with the corresponding expected outcome value provided by the tester and reports whenever it encounters a mismatch.

The above TAGL code is converted into the corresponding testing aspect code as shown in Listing 5.1. Based on the information provided in the TAGL denotation,

the required pointcut that captures the method to be tested, the around advice with the various values to be used for black box testing and method that compares the actual output with the expected output are created.

Listing 5.1: Generated testing aspect to test the *ComputeInterest* method of the *Banking* class

```
public aspect blackBoxTestComputeInterest {
        float around(float p,float r) : call(public float
            ↪ Banking.ComputeInterest(float,float)) && args(p,r)
        {
                float[] input= new float[] {0, 1, (float) 7.5, 10, 1000, -1,
                    ↪ 4096};
                for (int i = 0; i < input.length; i++)
                {
                        float actual_result = proceed(p, input[i]);
                        validateResult(i, actual_result);
                }
                return proceed(p,r);
        }

        void validateResult(int i, float actual_result)
        {
                float[] expected_output = new float[] {-1, 1000, 7500,
                    ↪ 10000, 1000000, 4096000};
                if(actual_result!= expected_output[i])
                {
                        System.out.println("Error at " + i + "th Input");
                        //Store in a suitable data structure for test report
                            ↪ preparation
                }
        }
}
```

Different input combinations can also be provided using our TAGL for testing the system under test. For example, in order to test a method of a *Triangle* class that determines the type of a triangle (returns 1 for isosceles, 2 for equilateral, 3 for scalene etc. and -1 if invalid inputs), it might be required to provide test cases with various input combinations for the length of the three sides of the triangle. Following is an example TAGL code that performs testing with such input combinations:

////type: blackbox
////aspectname: blackBoxTestGetType
////classname: Triangle
////methodsignature: static int GetType(int side1,int side 2,int side3)
////argumentname: (side1,side2,side3)
////values: (12,12,7),(-1,-1,-1),(0,0,0)
////expected: 1,-1,-1

Likewise, the tester can provide various type of combinations of inputs. For example, there could be a case where one of the input parameter has to be fixed and testing has to be performed by varying values of the rest of the input parameters. For example, if for testing the *GetType* method of the *Triangle* class as discussed above, the tester desires to keep the value of one input say *side1* fixed and further test with combinations of values for the other two inputs, then the following TAGL code can be used:

////type: blackbox
////aspectname: blackBoxTestGetType
////classname: Triangle
////methodsignature: static int GetType(int side1,int side 2,int side3)
////argumentname: side1=1,(side2,side3)
////values: (1,1),(-1,-1)
////expected: 2,-1

And for the same example, if the values of only one parameter are to be altered and the tester wants to keep the other two inputs at fixed values, then the following TAGL code can be used:

////type: blackbox
////aspectname: blackBoxTestGetType
////classname: Triangle
////methodsignature: static int GetType(int side1,int side 2,int side3)
////argumentname: side1=1,side2=1,(side3)
////values: -1,-32768, 1
////expected: -1,-1,2

Using our TAGL, it is possible to perform black box testing even with complex data types. The tester can provide different types of click events, keystrokes, voice, data files, tree, graphs or any other form of input as applicable to the

system under test. For example, testing a browser will require multiple Hypertext Markup Language (HTML) pages which can be provided using the same TAGL syntax as discussed above. Also an optional *////setup:* statement can be used for carrying out necessary set up activities (like setting values for particular data members of a class) before the method under test is executed. An example TAGL code is provided hereunder in which three methods namely, *setMarks1, setMarks2, setMarks3* are provided in the *////setup:* statement so that the marks of the student (*marks1,marks2,marks3* are data members of class *Student*) are set with given values before the *getAverage* method of the class is tested:

////type: blackbox
////aspectname: TestCaseMultipleInputs
////classname: Student
////methodsignature: public double getAverage()
////setup: public void setMarks1(int m), public void setMarks2(int m), public void setMarks3(int m)
////argumentname: marks1,marks2,marks3
////values: 4,5,6
////expected: 5

We indicated in Chapter 3 that TAGL simplifies the task of carrying out various types of black box testing like Fuzz Testing, Boundary Value Testing, All Pairs Testing, Orthogonal Testing, Equivalence Partitioning Testing etc. In the following subsections, we shall provide examples of TAGL denotations that can be used to create the corresponding testing aspects for these black box testing types.

5.4.1.1 TAGL for Creating Fuzz Testing Aspect

Fuzz testing, which is a type of black box testing and is carried out in order to find out how a system behaves when executed with insensible inputs, can be performed with ease using our TAGL. We wrote fuzzing aspect in Listing 3.16 of Chapter 3 which is used to inject some fuzz values into the text file associated with an application which reads input from such file. The following TAGL code can be used to generate such fuzz testing aspect automatically which fuzzes the input file and tests the response of the target application with the resultant abnormal input file:

////type: fuzztesting
////aspectname: fuzzTestReadFile
////classname: FileOperation
////methodsignature: public void ReadFile()
////filepath: "/home/administrator/inputdir/filename.txt"
////fuzzlocation: 5
////fuzzvalue: \%01\%02\%03\%04

In the above TAGL code, the first statement, as in the previous examples, indicates the type of testing to be performed. The following line which is optional specifies a name for the generated testing aspect and can be even omitted. Next statement specifies the class name which is to be tested and likewise the following statement specifies the signature of the method which reads the input file that has to be fuzzed. Then after the fully qualified path for the file which is to be fuzzed is provided using *////filepath:* . The next two statements *////fuzzlocation:* and *////fuzzvalue:* , which the tester specifies based on his experience, knowledge and the intent of testing, are the location in the file from where the input has to be fuzzed and the values that have to be used for such fuzzing.

Further, fuzzing of input file can be performed in various ways as we depicted in Figures 3.5, 3.6 and 3.7 of Chapter 3. These different types of fuzz testing can be carried out by introducing a new TAGL statement with item name *fuzztype* in the above TAGL denotation. In case of overwrite, the fuzz values provided by the tester are simply overwritten in the input file starting from the provided fuzz location. The TAGL code in the example above is simply the case of overwriting the fuzz value. For the case of overwriting, the value of *fuzztype* can be omitted or else it can be provided as *overwrite*. If no value of *fuzztype* is specified (as in the above example), then the corresponding testing aspect considers it to be the case of overwritten fuzzed values only.

Further, there could be case of insertion of fuzz values into the input file which can be done before or after a specific field of the file. In this case, the tester provides the *fuzztype* as either *insertbefore* or *insertafter* depending upon whether the insertion of the fuzz values has to be done before or after the specified field of the file. Moreover, both the starting and ending location of the field in whose neighbourhood the fuzz input has to inserted are also to be specified in a *fuzzlocation* statement separated by a comma. Following is an example of TAGL code whose corresponding generated testing aspect inserts the provided fuzz value before a given field:

////type: fuzztesting
////aspectname: fuzzTestReadFile
////classname: FileOperation
////methodsignature: private void ReadFile()
////filepath: "/home/administrator/inputdir/filename.txt"
////fuzztype: insertbefore
////fuzzlocation: 5,50
////fuzzvalue: \%01\%02\%03\%04

Also the particular contents of the file can be completely replaced by the provided fuzz values. In this case, the tester provides the *fuzztype* as *replace* and additionally, both the starting and ending location of the field of file to be replaced in the *fuzzlocation* statement separated by a comma. Example TAGL code for replacement of a specific portion of the input file is shown hereunder:

////type: fuzztesting
////aspectname: fuzzTestReadFile
////classname: FileOperation
////methodsignature: private void ReadFile()
////filepath: "/home/administrator/inputdir/filename.txt"
////fuzztype: replace
////fuzzlocation: 5,50
////fuzzvalue: \%01\%02\%03\%04

Moreover, here we would like to state that the generated testing aspect randomises the fuzz value provided in the *fuzzvalue* statement to create an unexpected invalid input and then injects it into the valid file as specified.

5.4.1.2 TAGL for Boundary Value Testing

Let us consider a *NumeralType* class which has got several methods that classify the input integer as positive/negative, prime/non-prime, palindrome/non-palindrome etc. Suppose we want to boundary test the *isPrime* method which takes an integer in the range [1,100] as input and outputs 0 if the number is not prime and 1 otherwise. The following TAGL denotation can be used to perform such testing with boundary values:

////type: boundaryvaluetesting
////aspectname: boundaryValueTestingAspect
////classname: NumeralType
////methodsignature: public int isPrime(int num)
////argumentname: num
////nominal: num=53
////values: (1,100)
////expected: (0,1,0,0,1)

In the above TAGL denotation, *////argumentname:* is used to specify the argument name for which boundary value test cases are to be generated. The following statement gives the allowed range for this argument. A new keyword *////nominal:* is introduced here which is used to provide the nominal value to be used while boundary value testing.

The *////expected:* statement in this TAGL provides the values for expected outputs which are matched with the actual outputs obtained from the boundary tests. Here we would like to state an important peculiarity of the testing aspect generated by the TAGL denotation of the type *boundaryvaluetesting*. The testing aspect creates an array of the boundary values for the given argument in a particular order viz. minimum value, a value just higher than the minimum value, maximum value, a value just lower than the maximum value and at last a nominal value as provided by the tester himself/herself. Thus, the tester should provide the expected output values in the TAGL denotation based on this order.

Likewise, the following TAGL denotation can be used to generate the testing aspect that tests the *returnDay* method of the *DateToDay* class as discussed in Section 3.2.1 of Chapter 3:

////type: boundaryvaluetesting
////aspectname: boundaryValueTestingAspect
////classname: DateToDay
////methodsignature: public String returnDay(int date, int month, int year)
////argumentname: date, month, year
////nominal: date=15, month=6, year=1967
////values: (1,31), (1,12), (1917,2017)
////expected: (Thursday,Friday,Friday,Invalid,Sunday,Wednesday,Wednesday, Friday,Friday,Saturday,Wednesday,Thursday,Thursday)

Table 5.2: Order of boundary value test cases in the auto-generated testing aspect

Date Value	**Month Value**	**Year Value**
1	6	1967
2	6	1967
30	6	1967
31	6	1967
15	1	1967
15	2	1967
15	11	1967
15	12	1967
15	6	1917
15	6	1918
15	6	2016
15	6	2017
15	6	1967

The generated testing aspect shall produce and execute the boundary test cases in a particular order as depicted in Table 5.2 which is self explanatory. The tester needs to provide the expected values accordingly.

Here we would like to state that the boundary value testing performed using TAGL assumes that the arguments whose boundary values are to be tested shall be of integer type only and thus is not suitable when the boundary value testing is to be performed with float or string type of arguments [70].

5.4.1.3 TAGL for All Pairs and Orthogonal Testing

TAGL can be used for all pairs testing as well. The following simple TAGL denotation is automatically converted by our lexical analyser and parser into the all pairs testing aspect that we showed in Listing 3.15 of Chapter 3:

////type: allpairstesting
////aspectname: allPairsTestingAspect
////classname: loanClass
////methodsignature: public boolean sanctionLoan(int no_of_kids,String occupation,boolean firstloan)
////argumentname: no_of_kids, occupation, firstloan
////values: (2,3,4),(Job,Business),(true,false)
////expected: *as per SRS*

The statements in the above TAGL denotation are similar to the previous TAGL denotations except for the *////type:* statement. Moreover, if the type statement in the above TAGL denotation is changed to *////type: orthogonaltesting*, a testing aspect with test cases based on the standard orthogonal array shall be generated by the TAGL which shall perform the orthogonal testing of the specified method.

5.4.1.4 TAGL for Equivalence Partition Testing

Equivalence partition testing is based upon classifying the input domains into valid and invalid and then selecting representative values from each partition as test cases. In order to understand how our TAGL simplifies the process of carrying out equivalence testing, let us take an example of a method *nextDate* in a class named *Calendar* which takes as input the current date and returns the next date. Let us assume that for this method, the possible values of year are restricted between 1917 to 2017. The following TAGL denotation is sufficient to generate a testing aspect to carry out the equivalence partition testing:

////type: equivalencepartitiontesting
////aspectname: equivalencePartitionTestingAspect
////classname: Calendar
////methodsignature: public String nextDate(int date,int month,int year)
////argumentname: date,month,year
////values: (1,31),(1,12),(1917,2017)

Following shall be the input range for this method:

$1 \leq \text{date} \leq 31$
$1 \leq \text{month} \leq 12$
$1917 \leq \text{year} \leq 2017$

TAGL heuristics partitions the input range of arguments into different equivalent classes in such a manner that testing with a particular representative value from one class is equivalent to testing of the other values from the same class. For example, TAGL partitions the input domain in above example into nine different equivalence classes (EC) in the following way:

$EC1 = \{1 \leq date \leq 31\}$
$EC2 = \{1 \leq month \leq 12\}$
$EC3 = \{1917 \leq year \leq 2017\}$
$EC4 = \{date < 1\}$
$EC5 = \{date > 31\}$
$EC6 = \{month < 1\}$
$EC7 = \{month > 12\}$
$EC8 = \{year < 1917\}$
$EC9 = \{month > 2017\}$

Based on the above 9 equivalence classes, our lexical analyser and parser prepares the test cases within the testing aspect such that a test case covers maximum valid input classes (EC1, EC2, EC3) and there exists a separate test case for each invalid class. The testing aspect generated from the above written TAGL denotation on the basis of test cases determined from these equivalence classes is shown in Listing 5.2.

Listing 5.2: Equivalence partitions testing aspect

```
public aspect equivalencePartitionTestingAspect {
        String around(int date,int month,int year) : execution(String
            ↪ Calendar.nextDate(int,int,int)) && args(date,month,year)
        {
                int[] dateTestValues = {15,0,32,15,15,15,15};
                int[] monthTestValues = {6,6,6,0,13,6,6};
                int[] yearTestValues = (1967,1967,1967,1967,1967,1916,2018};
                int i=0;
                String next_date="";
                for (i=0;i<dateTestValues.length;i++)
                {
                        next_date = proceed(dateTestValues[i],
                            ↪ monthTestValues[i],yearTestValues[i]);
                        //Store the output and the test case in a suitable
                            ↪ data structure for further analysis
                }
                return proceed(date,month,year);
        }
}
```

Here we would like to state a limitations of TAGL for the type *equivalencepartitiontesting* that the tester cannot provide the values for the expected outputs because the values of test cases chosen by TAGL heuristics is not known in advance. Therefore the generated testing aspect cannot compare the actual output

obtained with the expected output. However, the actual outputs obtained are recorded along with the equivalence partition's test cases in a data structure by the testing aspect which can be used by the tester to compare with the expected output as per the SRS visually. Also the TAGL assumes that the arguments whose domains are partitioned into equivalent classes shall be of integer type only.

5.4.2 TAGL for Creating Memory Leakage Testing Aspect

Java garbage collector (GC) takes care about the memory allocation and deallocation issues. Nevertheless, there are certain memory leaks which can still escape the GC. For example, if the programmer has written his/her own code for *finalize* method overriding the system's finalize() method in order to perform certain clean up duties before the object destruction and he/she forgets to call such method for the destruction of one or more of the objects created, then it shall lead to memory leakage. We suggested use of AOP to determine such leakage in Section 3.1.1 of Chapter 3 and provided example of AspectJ's aspect which we used for this purpose.

The tester can determine such memory leakage caused due to one or more forgotten call to the developer written *finalize* method by writing a straightforward denotation in TAGL. The following three simple TAGL statements shown hereunder are automatically converted into the memory leakage testing aspect by the TAGL lexical analyser and parser:

////type: countobjectstesting
////aspectname: testMemoryLeak
////classname: person

The generated testing aspect comprises of a static count variable and two different pointcuts to capture the creation and destruction of the objects of the provided class name and is similar to the aspect listed in Listing 3.1 of Chapter 3. Thus, it is apparent that a simple TAGL denotation, which is quite easy to learn and write, can be used by testers without the knowledge of AOP (AspectJ, in particular) to test their applications and find out the existence of undestructed objects.

One more type of memory leakage which occurs in Java programs is when we use object of a class, say *ClassB* within another class, say *ClassA* and it is not required for the complete life cycle of *ClassA*. We discussed this type of memory leakage in Section 3.1.1 of Chapter 3. TAGL can be used to determine such kind of a

memory leak which surfaces due to logical mistake of the programmer. Simple TAGL statements specifying only the type of testing, the name of the testing aspect that shall be generated (this statement is optional though), the name of the inner class (i.e. the class whose object has been created inside another class), method signature and the name of the outer class as shown hereunder are sufficient to generate the corresponding testing aspect:

////type: unusedobject
////aspectname: testMemoryLeakUnreferencedClassB
////innerclass: ClassA
////methodsignature: public void useB()
////outerclass: ClassB

The above TAGL denotation is converted into a testing aspect which has got a static counter and two pointcuts as shown in Listing 3.2 of Chapter 3 and which is capable of indicating a memory leak in case when the use of object of *ClassB* was finished before the life cycle of *ClassA*.

5.4.3 TAGL for Concurrency Testing

The following TAGL denotation can be used to generate the testing aspect shown in Listing 3.4 of Chapter 3:

////type: introducenoise
////aspectname: noiseInjectionAspect
////threadname: Division
////insertnoise: after
////probabilitypercentage: 1
////sleep: Random(35)

In the above TAGL denotation, *////threadname:* statement provides the name of the thread under consideration. The following statement is used to indicate before/after, i.e., with respect to the thread run, when the noise has to be injected. The next statement regarding probability of noise injection is optional and is used to implement the heuristics associated with noise injection. With the above example denotation, noise is inserted with a probability of 1%. *////sleep:* in the last statement is used to indicate how long the thread has to be made to sleep for

testing. It can be used to provide a random value as shown in this denotation or even a fixed integer sleep duration (like *////sleep: 35*).

Likewise, the following TAGL denotation can be used to generate the testing aspect which is used to insert heuristic noise after every shared variable access as shown in Listing 3.5 of Chapter 3:

////type: introducenoise
////aspectname: noiseInjectionAspect
////threadname: Shared
////sharedvariable: a
////insertnoise: after
////sleep: Random(20)

The *////sharedvariable:* statement is used to provide the name of the shared variable. Other statements are similar to the previous TAGL denotation shown above.

5.4.4 TAGL for Creating Null Pointer Exception Checking Aspect

When a programmer uses a reference that points to a null location in the memory, a null pointer exception is raised. Such a situation usually arises when the programmer refers to a null object for calling a method of the class or for accessing a field value of such object. Following straightforward TAGL denotation can be used by the tester for testing the complete source code of an application for undesirable calls to any method or setting of a data member of a particular class using an uninitialized null object of that class, anywhere within the code:

////type: nullpointerexception
////aspectname: testNullPointerException
////classname: testClass

The above TAGL denotation is converted into a testing aspect with a pointcut that captures calls to all the methods and setting of all the data members of the class and further uses the *target* pointcut to capture the execution object. If the execution object is null, the AspectJ construct *thisjoinpoint* captures the context information regarding its location. This context information happens to

be useful for the tester to detect the cause of the null pointer issue. The code of the generated testing aspect is shown in Listing 5.3.

Listing 5.3: Testing for unhandled null pointer exceptions

```
public aspect testNullPointerException {
        pointcut NullPointerCheck(testClass obj) : (call(* testClass .*(..) )
            ↪ || set(* testClass .*)) && target(obj);
        before(testClass obj) : NullPointerCheck(obj)
        {
                if(obj==null)
                {
                        System.out.println("Null pointer exception in: " +
                            ↪ thisJoinPoint + " at: " +
                            ↪ thisJoinPoint.getSourceLocation());
                        System.exit(0);
                }
        }
}
```

5.4.5 TAGL for Creating Load Testing Aspect

Load testing entails creating dummy users or requests and testing the application with these to get an insight how the application under test will perform under load conditions. TAGL is useful for the load testers for testing the performance of Java applications under dummy load. In the TAGL denotation for load testing, the *////type:* has to be specified as *loadtesting*. The class whose multiple objects the tester wants to create as dummy users followed with the number of such objects have to be mentioned using item names *////classload:* and *////numberofobjects:* respectively. In the example given in Listing 3.21 of Chapter 3, we have load tested a shopping cart application with multiple users. In particular, the *addUser* function of the *shopping* class has been tested with 1000 number of users. The following TAGL denotation can be used to generate the same load testing aspect:

////type: loadtesting
////aspectname: testShoppingCart
////classname: shopping
////methodsignature: public void addUser(shopping s)
////classload: shopping
////numberofobjects: 1000

Similarly, the following TAGL can be used to perform load testing of the shopping cart application with 5000 number of items shopped by a single user:

```
////type: loadtesting
////aspectname: testShoppingCart
////classname: User
////methodsignature: public int shop(Item i)
////classload: Item
////numberofobjects: 5000
```

The above simple TAGL denotation can be used to generate the testing aspect as shown in Listing 3.22 of Chapter 3 which creates 5000 objects of the *Item* class and calls the *shop* method of the *User* class.

There are testing scenarios when before calling the method to be load tested with the created dummy objects, the dummy objects so created are required to be initialised or in other words, certain data members of such dummy objects have to be *set* before they can be put to use for the purpose of load testing. In such cases, the required initialisations should be specified in the form of Java statements using *////loadtestinginitialsetup: Java statements*. One example TAGL code, where the *id* and *salary* of the objects of the *Employee* class are to be set before using them for the load testing of the *CalculateIncomeTax* method is shown hereunder:

```
////type: loadtesting
////aspectname: testCalculateIncomeTax
////classname: Employee
////methodsignature: public void CalculateIncomeTax()
////classload: Employee
////numberofobjects: 1000
////loadtestinginitialsetup:
{
      Random rn = new(Random);
      for(i=1;i<=1000;i++)
      {
         e.id=i;
         e.salary = rn.nextInt(10000,100000);
      }
}
```

The above TAGL code generates a load testing aspect which creates one thousand dummy objects of the class *Employee* and sets each object's *id* and *salary* before calling the *CalculateIncomeTax* method that has to be load tested. Thus, in this way TAGL can be used to carry out the initial set up that is required to be done before using the dummy objects for load testing.

5.4.6 TAGL for Creating Servlet Testing Aspect

Servlets in Java are used for generating dynamic content on the Web and have native support for the Hypertext Transfer Protocol (HTTP). In servlet programming, input from the user that is entered in fields like textbox, combobox etc. on the HTML page is forwarded to the servlet which further stores this information into the database or does other necessary processing as per the context.

In the Section 3.3.2 of Chapter 3, we explained how AspectJ can be used for performing servlet testing. We used aspects to capture the desirable execution points within the servlet to be tested and for this purpose we have implemented the *Filter* interface. Further, we have overridden the *getParameter()* method to pass suitable test parameters to the servlet for the purpose of security testing. Such servlet testing aspect can be easily generated with a simple denotation using our TAGL. We only need to specify the name of the HTML form parameter that we want to test by providing different input values and further the various values that have to be used for the servlet testing. Following is a sample denotation to understand the TAGL syntax for servlet testing:

////type: servletesting
////aspectname: testServlet
////parametersname: username
////values: "1;DROP TABLE users","' OR '1'='1"

In the above TAGL for servlet testing, the *////parametersname:* statement provides the name of the form parameter that has to be tested. This example denotation is used for testing the *username* input text box. The *////values:* statement is used to provide the various values that have to be used for testing the vulnerabilities of the servlet. In this example TAGL denotation, we have used two SQL values that are injected in the *username* form parameter. These SQL test cases checks for the existence of incorrectly filtered escape characters left by the developer in the code. The first test case manipulates the SQL query in the

servlet code in such a way that an additional query to drop the users table (if such table exists) is run. And the second test case simply returns all the records as it appends the insecure query written by the programmer with an *always true condition*.

Tester might require to provide combination of values for different form parameters at a time; for example when a particular servlet has to be tested for combination of *(username,password)* values. TAGL can also be used in such a case where values of two form parameters are to be provided simultaneously. Following sample denotation spells out the TAGL syntax:

////type: servletesting
////aspectname: testServlet
////parametersname: (username,password)
////values: (" OR ""="," OR ""=")

The above TAGL denotation is quite similar to the syntax explained earlier for the testing of servlet with different values for one form parameter, only except for the fact that we have provided names of two parameters and their values for testing has been provided within brackets to be used as one combination, i.e., a single test case. The input combination provided in the example TAGL syntax will manipulate the following insecure query written by the developer and fetch all rows from the table of users since *OR ""="" is always true*:

'SELECT * FROM usertable WHERE Name ="' + userName + '" AND Password ="' + password + '"'

Likewise, the following TAGL can be used to test the servlet shown in Listing 5.4 that processes two form parameters, *username* and *password*, and stores them into *mysql* database, with *null* values for both the parameters:

////type: servletesting
////aspectname: testServlet
////parametersname: (username,password)
////values: (null,null)

The generated testing aspect is shown in Listing 5.5. It is used to test the servlet code and determine whether the developer has handled the case of *null* values in the servlet code or not.

Listing 5.4: Servlet with two form parameters

```
@WebServlet("/DBServlet")
public class DBServlet extends HttpServlet {
        private static final long serialVersionUID = 1L;

        public DBServlet() {
                super();
        }

        protected void doGet(HttpServletRequest request,
            ↪ HttpServletResponse response) throws ServletException,
            ↪ IOException {
                doPost(request, response);
        }

        protected void doPost(HttpServletRequest request,
            ↪ HttpServletResponse response) throws ServletException,
            ↪ IOException {
                response.setContentType("text/html");
                response.setHeader("Cache-Control", "no-cache");
                //get params
                String userName=request.getParameter("username");
                String passwrd=request.getParameter("password");
                try {
                        //Load the database driver
                        Class.forName("com.mysql.jdbc.Driver");
                        java.sql.Connection con =
                            ↪ DriverManager.getConnection("jdbc:mysql://
                            ↪ localhost:3306/test", "admin", "******");
                        PreparedStatement ps = con.prepareStatement("insert
                            ↪ into Customer values(?,?)");
                        ps.setString(1, userName);
                        ps.setString(2, passwrd);
                        int result = ps.executeUpdate();
                        //send response as result
                        response.getWriter().write(Integer.toString(result));
                }
                catch (Exception e2) {
                        System.out.println(e2);
                }
        }
}
```

Listing 5.5: Aspect for testing servlet with two form parameters with *null* values

```
public aspect testAspect implements Filter
{
        public void doFilter(ServletRequest request,ServletResponse
            ↪ response,FilterChain chain) throws
            ↪ IOException,ServletException
        {
                chain.doFilter(new
                    ↪ RequestWrapper((HttpServletRequest)request),response);
        }
        @Override
        public void destroy()
        {
        }
        @Override
        public void init(FilterConfig arg0) throws ServletException
        {
        }
}
public class WrapperRequest extends HttpServletRequestWrapper {
        public WrapperRequest(final ServletRequest request) {
                super((HttpServletRequest) request);
        }
        @Override
        public String getParameter(final String name) {
                if(name.equals("username")) {
                        return null;
                }
                //When the request(string) has got password
                else {
                        return null;
                }
        }
}
```

5.5 Lexical Analyser and Parser

In order to convert the TAGL denotations written by the testers into the testing aspects, we have written lexical analyser and parser using *lex* and *yacc* tools. The lexical analyser written in lex reads the TAGL statements and breaks them into tokens. The parser written using yacc receives these tokens, imposes the grammar rules defined in the yacc file and takes necessary actions to generate the testing aspects as specified. The process of automatic aspect generation from TAGL input

using lex and yacc is depicted in Figure 5.3. *yylex()* is the name provided by lex to the main entry point for the lexical analyser generated by it. *yyparse()* is the function created by yacc which causes parsing to occur.

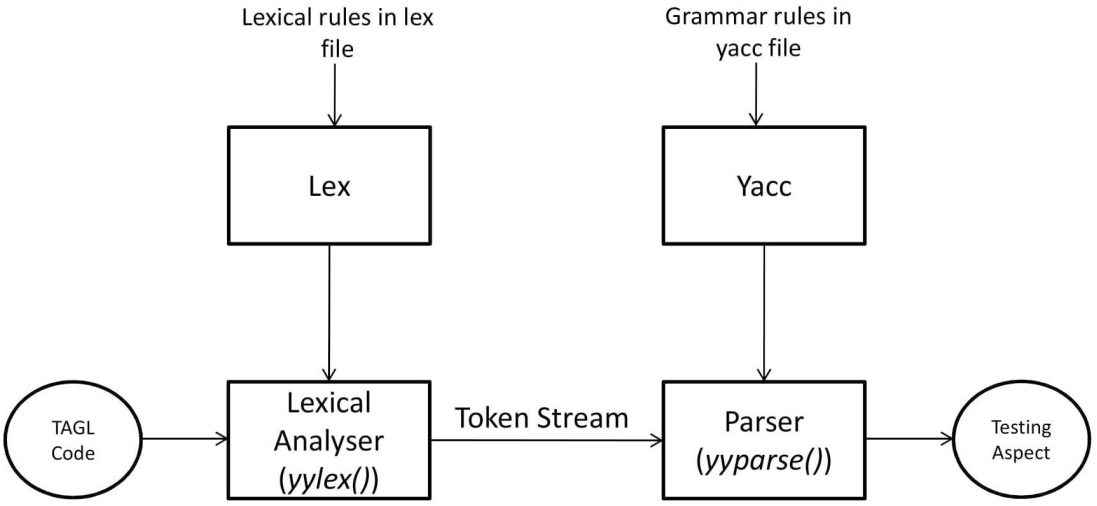

Figure 5.3: Automatic conversion of TAGL into testing aspect using lex and yacc

The TAGL statements to be converted into testing aspects can be written within the original source code of the system under test (in the same way the developer writes the program comments) or in separate files, if so desired. As all the statements in our TAGL start with 4 backslashes "////", the Java compiler considers them as comments and are thus not executed but these are still distinguishable by our lexical analyser and parser while producing the testing aspects. As per the rules specified with regular expressions in the lex program and the grammar in the yacc program, a new TAGL statement is said to be detected whenever a "////" is encountered in the TAGL denotation.

If the TAGL statements have been embedded within the original source code of the system under test, then the complete source code file is parsed by our lexical analyser and parser and whenever it finds a TAGL denotation, it creates the corresponding testing aspect from it. If the TAGL statements have been specified in a separate file, then such file is read by our lexical analyser and parser in order to generate the testing aspects. Here we would like to highlight that at least one blank line gap is necessary to be left between two consecutive TAGL denotations.

The lexical analyser has been prepared using the lex tool which reads the strings in the TAGL statements and produces meaningful tokens based on the rules specified in the form of various regular expressions. Important tokens that have been defined in the lex program along with their descriptions are listed in Appendix B. Our

lexical analyser produces these meaningful tokens from the TAGL statements and communicates to the parser.

An example on how the lexical analyser reads the TAGL statements and generates relevant tokens for the parser is shown in Listing 5.6. In this lexical rule which is defined within our lex program, whenever a primitive type like int, float, double etc. is encountered in the TAGL denotation, it is judged whether it is return type of a method or the type of its arguments. Accordingly a token RETURNTYPE or METHODARGTYPE is passed to the parser. The statement *yylval.string=strdup(yytext)* is meant to pass the name of the primitive type viz. int, float, double etc. to the parser.

Listing 5.6: Example lexical rule from lex program

```
byte|short|int|long|float|double|boolean|char|void        {
        if(state==methodsignature && startmakingargumentlist==OFF)
        {
                yylval.string=strdup(yytext);
                return RETURNTYPE;
        }
        /*whenever a opening bracket is encountered in the method
            ↪ signature,startmakingargumentlist is turned ON*/
        if (state==methodsignature && startmakingargumentlist==ON)
        {
                yylval.string=strdup(yytext);
                return METHODARGTYPE;
        }
}
```

Further, we specify the TAGL rules and the code to be invoked when these rules are recognised in the grammar in the yacc program. These rules, which are commonly called *production rules*, describe the allowable structures in TAGL. The grammar is read by the yacc tool to produce a *Look-Ahead LR parser* (LALR) parser. When one of the rules specified in the parser has been recognised, then the code supplied in the action part for this rule is executed and all such actions combined together generate the testing aspect from the TAGL statements. For example, following are the grammar rules starting with the *start symbol S* that match the TAGL denotations for performing memory leakage testing (////type: countobjectstesting) as well as null pointer exception checking discussed in Section 5.4:

S : MLTANDNPETDENOTATION

Further the rule for MLTANDNPETDENOTATION is shown hereunder:

MLTANDNPETDENOTATION : ASPECTTYPESTMT ASPECTNAMESTMT CLASSNAMESTMT

In the above grammar rule, the non-terminal ASPECTTYPESTMT corresponds to the *type* statement in the TAGL denotation which is used to specify the type of testing to be performed. The non-terminal ASPECTNAMESTMT corresponds to the name to be used for the generated aspect. And the non-terminal CLASSNAMESTMT corresponds to the class for which memory leakage has to be checked or whose uninitialized null objects are to be explored.

A snippet of grammar rule, that is meant for matching a method signature provided by the tester in the TAGL denotation, as defined in our yacc file is shown in Listing 5.7.

Listing 5.7: Yacc grammar snippet for matching a method signature

```
METHODSIGNATURE : ACCESSSPECIFIER RETURNTYPE
    ↪ METHODNAME METHODARGLISTS
                        {
                        strcpy(methodreturntype,$2);
                        strcpy(methodname,$3);
                        }
                ;
METHODARGLISTS :
                |       METHODARGLISTS METHODARGLIST
                ;
METHODARGLIST :   METHODARGTYPE METHODARGNAME
                        {
                        strcpy(funcArgTypeTable
                            ↪ [funcArgSymbolTableIndexY], $1);
                        strcpy(funcArgSymbolTable
                            ↪ [funcArgSymbolTableIndexY], $2);
                        funcArgSymbolTableIndexY++;
                        }
                ;
```

Here, we would like to state that there are separate grammar rules for all the non-terminals in the yacc program. These rules are matched as and when the terminals are received from the lexical analyser. Terms that appear on the left hand side of a grammar rule, like METHODSIGNATURE, are non-terminals. Terms such as

ACCESSSPECIFIER or RETURNTYPE are terminals which appear on the right hand side of a grammar rule. Terminals represent the basic symbols of which the language is composed of. Terminals are compared to the grammar rules at every step and when one of the rules is recognised, the action code supplied for the rule is invoked. Further, the value held by the ith symbol in a rule can be accessed using "$i".

In order to get a complete insight into how the yacc parser actually works, we are providing snippets of our yacc program that suffice for producing a memory leakage testing aspect in Listing 5.8, 5.9 and 5.10. The source code for the *WriteAspectnameInAspectFile* and *WriteClassnameInAspectFile* methods which are called in the action part of the grammar rules written in Listing 5.8 are shown in Listing 5.9 and Listing 5.10 respectively. **The complete grammar of our TAGL has been provided at Appendix C.**

Listing 5.8: Yacc grammar snippet for generating memory leakage testing aspect-I

```
S       :       ASPECTTYPESTMT ASPECTNAMESTMT
    ↪ CLASSNAMESTMT
        ;
ASPECTTYPESTMT : COMMBLOCK TYPE COLON ASPECTTYPE
                        {
                        strcpy(aspecttype,$4);
                        }
                        ;
ASPECTNAMESTMT : COMMBLOCK NAME COLON ASPECTNAME
                        {
                        WriteAspectnameInAspectFile($4);
                        }
                        ;
CLASSNAMESTMT :  COMMBLOCK CLASSNAMETAG COLON
    ↪ CLASSNAME
                        {
                        strcpy(classname,$4);
                        if(strcmp(aspecttype,"countobjectstesting")==0)
                                WriteClassnameInAspectFile($4);
                        }
                        ;
```

Listing 5.9: Yacc grammar snippet for generating memory leakage testing aspect-II

```
void WriteAspectnameInAspectFile(char *aspectname)
{
        char fileName[100];
        strcpy(fileName,aspectname);
        strcat(fileName,".aj");
        fp=fopen(fileName,"w");
        fprintf(fp,"public aspect ");
        fprintf(fp,"%s\n{\n",aspectname);
}
```

Listing 5.10: Yacc grammar snippet for generating memory leakage testing aspect-III

```
void WriteClassnameInAspectFile(char *classname)
{
        fprintf(fp,"static int count=0;\n");
        fprintf(fp,"pointcut creation(%s obj) : execution(public ",classname);
        fprintf(fp,"%s",classname);
        fprintf(fp,".new(..)) && this(obj);\n");
        fprintf(fp,"after(%s obj) :
            ↪ creation(obj)\n{\ncount++;\n}\n",classname);
        fprintf(fp,"pointcut destruction(%s obj) : execution(protected void
            ↪ ",classname);
        fprintf(fp,"%s",classname);
        fprintf(fp,". finalize (..) ) && this(obj);\n");
        fprintf(fp,"after(%s obj) :
            ↪ destruction(obj)\n{\ncount--;\n}\n",classname);
        fprintf(fp,"after() : execution(public static void main(..))\n{\n");
        fprintf(fp,"if(count>0)\n{\n");
        fprintf(fp,"System.out.println(\"Memory Leak!\");\n}\n");
        fprintf(fp,"else\n{\n");
        fprintf(fp,"System.out.println(\"No Memory Leak!\");\n}\n");
        fprintf(fp,"}\n");
        fprintf(fp,"}\n");
}
```

5.6 Summary

There are different patterns of the TAGL statements based on the grammar rules for performing different types of testing but all of them have got obvious syntax that is quite natural language-like and thus easy to learn. Testers without the knowledge of AOP can still reap the benefits of using AOP for software testing with the help of our easy to write TAGL statements which are automatically converted into testing aspects by the lexical analyser and parser that we have developed using lex and yacc. The friendly learning curve of our TAGL allows the tester to get started quickly and instantly perform productive testing.

Chapter 6

Comparison with Conventional Technologies:Qualitative Analysis

Growing dependency of mankind on software technology increases the need for thorough testing and automated techniques that support testing activities. In our research work, we have outlined a novel testing strategy for performing various types of software testing using Aspect Oriented Programming. Further, we have developed a Testing Aspect Generator Language (TAGL) that can be used by the testers without the knowledge of AOP for writing test scripts in the form of natural language like statements. In this Chapter, we shall asses the usefulness of our proposed approach by comparing it with the existing testing methodologies. We shall compare our approach with the conventional techniques on the basis of functionality, ease of use, learning curve, flexibility, aid for bug resolution, lines of testing code, code coverage, test execution times, types of testing covered, support for complex data types and non functional requirements etc. A complete qualitative analysis (in this Chapter) as well as quantitative analysis (in the next Chapter 7) shall be carried out in order to evaluate our proposed AOP approach for performing various types of software testing and establish the benefits thereof. We shall also discuss the effectiveness of our approach when applied to real world software applications and elucidate for what all type of programs shall our proposed approach be best suited. Substantially we have used AspectJ for the purpose of comparison as it has become the de-facto standard for AOP.

6.1 Resemblance with JUnit: most popular testing tool for Java applications

Automated testing tools simplify the testing efforts because the tester has to write the minimum test script and repetitive execution of test cases is simplified. Also automated testing tools provide for means of comparing the actual output with the expected output. In a nutshell, automated testing tools make the testing process faster and more efficient. For testing Java applications, JUnit is the most popular and extensively used open source automated testing tool which targets individual methods and classes [55, 71, 72].

AspectJ, the AOP extension for Java, has become the de-facto standard for AOP by emphasising simplicity and usability for end users [73]. Our proposed AOP approach for testing Java applications using AspectJ has got profound resemblances with JUnit on many facets and as such covers all sort of testing functionality provided by JUnit. In JUnit, the tester creates the *test classes* within which the testing code is written. Similarly when using our approach for testing, the tester writes the testing code within the aspects in AspectJ which are quite class-like concept [74]. Aspects in AspectJ behave like Java classes with the difference that these can have pointcuts and advices within them. Class is the unit of modularity in Object Oriented Programming and aspect is the basic building block in Aspect Oriented Programming that modularises the concerns.

JUnit provides with annotations which are like meta-tags that can be added to the testing code. JUnit annotations are meant to identify when or in which order the various methods in the test class are to be executed. For example, the *@Test* annotation is used to specify the test method that has to be run as test case. In AspectJ, the same functionality is achieved by the *around* advice in which the method to be tested can be instrumented and tested with desired input values. This analogy has been depicted in Listing 6.1 and 6.2.

The annotations *@Before* and *@BeforeClass* in JUnit are used to indicate the methods which setup the necessary pre-conditions required for the execution of the test methods. The *@Before* annotation is used with a method that has to run before every test case in the test class. We have *before* advice in AspectJ that serves the same purpose like *@Before* annotation as evident from Listing 6.1 and 6.2. Code written within a before advice with appropriate pointcuts that capture the methods to be tested shall be executed before the testing code written within the around advices. Similarly, the method marked with annotation *@BeforeClass*

in JUnit is executed once before the test class. In these example listings, the marks of the student in three subjects are set using the *@Before* annotation in JUnit class *TestCase* and using the *before* advice within the aspect *testAspect* before the method that calculates the average can be tested.

Listing 6.1: Testing a method in *Student* class using JUnit

```
import static org.junit.Assert.*;
import org.junit.*;

public class TestCase {
        Student s = new Student();

        @Before
        public void doBefore()
        {
                s.setMarks1(5);
                s.setMarks2(6);
                s.setMarks3(7);
        }

        @Test
        public void test()
        {
                double avg = s.getAverage();
                assertEquals(6.0, avg, 0);
        }
}
```

When a JUnit test class comprises of multiple methods which are marked with *@Test* annotation and which all require certain necessary precondition to be setup, then a method which sets up the pre-condition marked with "@BeforeClass" annotation is used. To achieve the same functionality in AspectJ, we used the *adviceexecution* pointcut within an aspect which captures the execution of all the advices within the testing aspect. Further a before advice was used to execute the desired pre-condition setup code before the execution of advices in the testing aspect as shown in Listing 6.3. In this example listing, the static variable i has been used to ensure that the setup code executes exactly once.

Likewise, the annotations *@After* and *@AfterClass* are used to indicate the methods which get executed after execution of the tests methods and perform certain cleanup tasks like delete temporary variables, reset variable, disconnect from database etc. These JUnit annotations can be directly mapped onto the *after* and

Listing 6.2: Testing a method in *Student* class using AspectJ

```
aspect testAspect
{
        Student s = new Student();
        before() : execution(public double Student.getAverage())
        {
                s.setMarks1(5);
                s.setMarks2(6);
                s.setMarks3(7);
        }

        double around(Student st) : execution(public double
            ↪ Student.getAverage()) && this(st)
        {
                double x = proceed(s);
                if (x!=6) System.out.println("Error: expected output: 6 and
                    ↪ actual output: "  + x);
                return proceed(st);
        }
}
```

Listing 6.3: Aspect equivalent to the *@BeforeClass* annotation in JUnit

```
public aspect beforeTestingAspect {
        static int i = 0;

        pointcut beforeAll() : adviceexecution();
        before() : beforeAll() && !within(beforeTestingAspect)
        {
                if(i++ == 0)
                {
                        //Do the desired pre-condition setup here
                }
        }
}
```

adviceexecution advices available in AspectJ along with appropriate pointcuts, on the same lines as discussed for *@Before* and *@BeforeClass*. Suitable code to perform the desired reset or release of resources is written within these *after* and *adviceexecution* advices.

Sometimes while running a test class the source code under test is not completely ready. The *@Ignore* annotation in JUnit is meant to ignore the execution of a particular test method. Methods annotated with @Test that are also annotated with @Ignore are not executed as tests. The same functionality can be achieved in AspectJ by adding a simple "&& if(false)" to the pointcut so that the corresponding advice shall not be executed [18]. The example code snippet in Listing 6.4 shows the syntax for nullifying an advice.

Listing 6.4: Ignoring the execution of a testing advice alike JUnit

```
pointcut selectedJoinpoints() : within(package.*) && if(false);
before() : selectedJoinpoints ()
{
        //Testing operations (these shall not be executed)
}
```

As discussed above, all the annotations in JUnit can be equated with one of the available constructs in AspectJ. Thus, the functionality of JUnit and the types of testing of Java applications that can be carried out using JUnit are equally possible with AspectJ.

6.2 Advantages of the proposed AOP and TAGL approach

Furthermore, there are several advantages of using AspectJ over JUnit for performing software testing that we shall discuss hereunder.

6.2.1 Learning Curve

Although there are several benefits of using automated testing tools but still new costs surface up like the implementation and training costs. Moving to test automation means that the way of working of the testers shall change and definitely

training on the automated testing system would be required. The most important among the reasons for failed automation is the inexperience of testing team with the automation tool and thus proper training of the tool is required [75]. Moreover, before such training it is important to convince testers that the automation of the manual practises shall bring real benefits and will not be much additional burden [76].

Douglas Hoffman [77] states that *Writing of automated tests is more difficult than manual tests and requires a super set of knowledge and experience over manual testing. Not all members of an existing test group are able to make such changes.* There are various tools that help software teams build and execute automated tests but the team is required to acquire proficiency in writing the test scripts using the tool. For example, for tools like JUnit, TestNG, Selenium etc. meant for testing Java applications, the tester is first required to learn them before starting to use. In a nutshell, the biggest drawback for all the automated testing tools is that these increase the expected skill-level from the testers. Thus for improving the testing process, it is important to improve the automation tool and make it easier to learn and use.

We have proposed AOP for the purpose of automating the process of software testing. AspectJ, which is the most popular and in-practise language for AOP, is easy to learn and use [78, 79]. AspectJ's syntax is quite similar to that of Java except for the new constructs meant for capturing the crosscutting concerns and thus Java application testers can pursue its use for testing without putting in too much efforts. Further to instigate a new level of abstraction, we have developed our *Testing Aspect Generator Language (TAGL)* which can be used by Java application testers for delineating the test cases to test their applications. TAGL statements are converted into AspectJ testing aspects automatically by the lexical analyser and parser that we have written using lex and yacc. As we explained in Chapter 5, the syntax of our TAGL is quite intuitive and thus the testers can quickly learn TAGL and start writing the testing code in natural language-like TAGL statements.

In order to evaluate the usability of our TAGL, we selected 30 students from the final year batch of an engineering college who were novice in the field of testing and had little or no knowledge of the testing tools and techniques. We made them to learn both JUnit and our TAGL and then test several Java programs in the laboratory. As the students were not skilled in both the techniques, they had to first acquire the level of proficiency to start writing the testing code using these

techniques. The basic programs included for the purpose of testing comprised of sorting programs, stack and queue classes, calculator, simple servlet programs, banking and library applications developed by the students themselves during their curriculum.

After a regular learning and usage of 30 days, the students were asked to rate both the techniques. Following is a summary of the observations:

- Ability to meet the objectives: As per the student tester's experience, both TAGL and JUnit could meet the primary objective of software testing; however, TAGL could cover many different types of testing. They could detect bugs related to memory leaks, interference and also perform load testing using TAGL whereas with JUnit, they could only perform testing of the modules with the input test data and use *assertion* methods for correctness.

- Ease of learning: All the 30 students commended that TAGL has got quite natural-language like syntax and thus is far easy to learn as compared to JUnit. The same is evident from Figure 6.1 which shows the learning curves of these students for TAGL and JUnit.

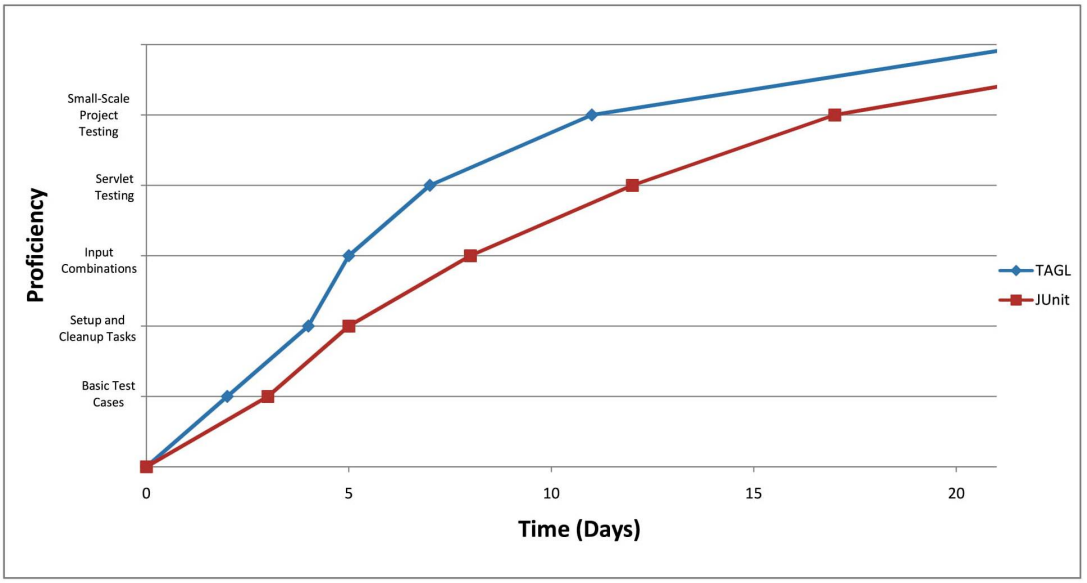

Figure 6.1: Learning curve of novice testers for TAGL and JUnit

- Memorability: Memorability refers to the property of an artifact that measures how easy is it to remember when a user returns to the artifact after not

using it for a certain period of time. We made the student testers to return and use the testing techniques after a period of not using it for 7 days, 83% of them remembered our TAGL syntax sufficient enough to use it effectively for the next time. It is so because we have kept the TAGL syntax instinctual and the constructs of the language quite natural-language like. However, in case of JUnit, 60% of them had to start over again to learn everything.

- Ease of use: Ease of use refers to the extent to which a user can use an artifact to achieve specified goals with satisfaction. When the students wanted to test their programs with multiple input value combinations simultaneously, they had to use the *Parameterized class* in JUnit which they found quite complex whereas they found it simple to provide multiple inputs using TAGL. Overall, the natural language like syntax of TAGL makes it easy to use for novice as well as experienced testers.

- Time taken for writing the code: Time taken by the students for writing the testing code using JUnit is more because the syntax involves use of various annotations and punctuations.

- Error messages: Regarding error messages generated upon wrong statements in testing code, most of the students advocated that JUnit provides better and more meaningful error messages which are helpful in debugging the testing code. TAGL falls weak on this perspective. However, the number of syntax errors made by the student testers was lesser with TAGL because the syntax is quite natural-language like and uncomplicated to write test code with.

- Test reports: Test report shows the comparison of test results with expected results. From the student's experience, reporting of failures by our technique was simple and easy to understand. Although the test reports produced by JUnit were more informative but they happen to be complex.

- Task completion rate: Testing tasks given to students included fault injection testing in deposit and withdraw modules of banking application, testing of the calculator program like divide by zero or negative number, testing of bubble sorting program with different input combinations like empty list, repeated numbers in list etc., testing the push and pop operations of stack,

testing for stack overflow, testing operations of a circular queue. Additionally TAGL specific testing tasks were also assigned like memory leakage testing of library application's classes, interference testing of the parallel sorting programs etc. which cannot be performed using JUnit. After being reasonably trained on TAGL as well as JUnit, on an average 93% students could complete the given tasks successfully using our TAGL. However, the mediocre students faced difficulties in using JUnit and on an average only 16 of the total 30 students could accomplish the given testing activities. (Note: such average was calculated as the summation of number of students that performed each task divided by the total number of tasks)

6.2.2 Modification of source code for testing

For carrying out certain types of testing like non-functional testing, memory leakage testing, invariant testing, interference testing etc., the source code is required to be modified [12, 80]. The tester need to bring about changes in the original source code and the testing code has to be written in various modules.

Various types of software requirements and the level at which the non-functional requirements come into picture are depicted in Figure 6.2. Non functional requirement of a software are those which determine *how* a software will function (and not *what* the software will do). Non functional requirements describe the performance constraints, external interfacing conditions, scalability and security needs, robustness etc. These are catalogued by the people of technical know-how like the developers or team leaders. Despite the fact that lot of work has been carried out that emphasises the importance of the non-functional requirements, the testing approaches available for the evaluation of non-functional requirements are not many [11].

Non-functional properties of software are realised by code that is spread across the complete application. Thus, testing for such requirements is either not possible with tools like JUnit or else entails scattering of testing code at number of places within the application's original source code when specifically designed test tools are used. For example, there could be a non-functional *profiling* requirement of an application that the response time of various procedures to service requests should be within specified time limits. For this it would be required to track the executions of various procedures and prepare execution time profiles. For testing for such requirement, it would be required to insert testing code (to calculate functions'

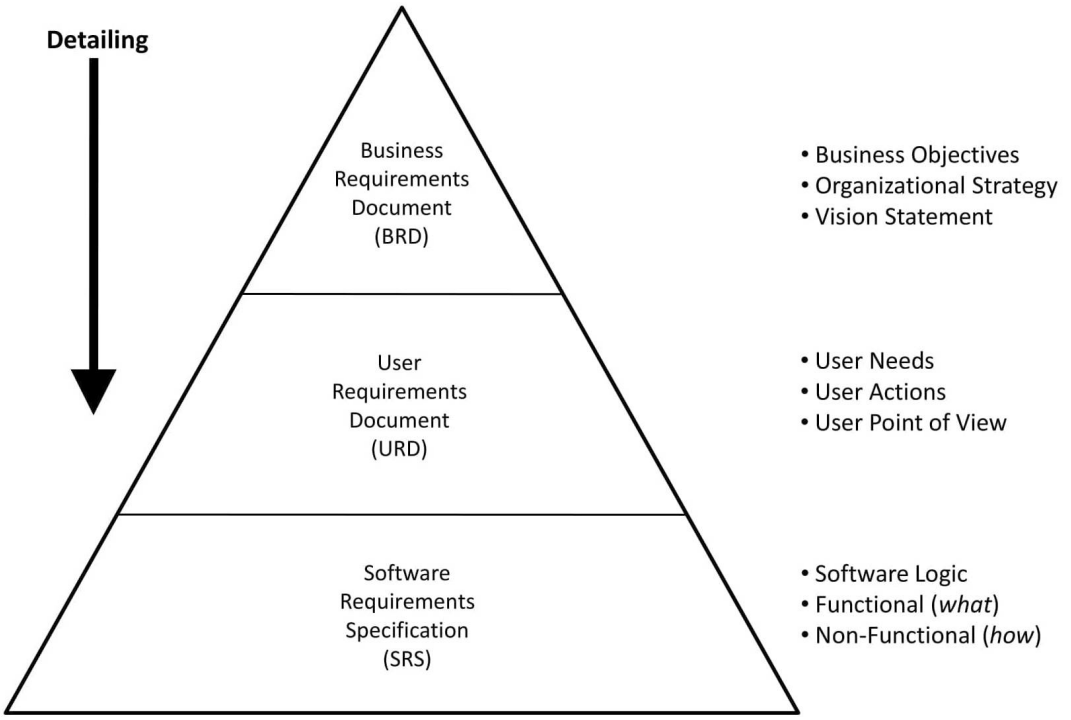

Figure 6.2: Types of software requirements and corresponding documents

execution times) at several places, namely before and after the execution of every function. However, using aspects we can capture the execution of all the functions by using a wild card pointcut within a profiling aspect and then further calculate the execution times using a before and after advice. Similarly, we can check for the memory footprint of various modules using aspects without modifying the source code to test that the memory consumption is within the allowed limits.

Likewise if we talk about *robustness* which is yet another non-functional requirement and thus testing for it shall involve substantial instrumentation of the system under test. However, using AOP we can write aspects that contain the code for simulating users or connections to test for robustness. To illustrate, lets take an example of a servlet which opens a database connection but does not close it and relies upon the Java garbage collection for release of the database objects. As the number of connections to a database could be limited, an attacker can make use of this vulnerability to create multiple servlet calls and bring about a situation of denial of service when the application is overburdened with numerous live database connections. Aspects in AspectJ can be used to test the application for robustness against denial of service by loading it with multiple calls to the servlets.

Moreover, it is equally important to mention here that modifications to source code for the purpose of testing may lead to maintenance issues as well as surprises

for the end user of the application if the developer or tester forgets to delete these. Using AOP for testing rescues the developers and testers from worrying about these issues.

6.2.3 Testing Private Members

JUnit does not provide upfront mechanism for testing the private methods. As private methods can only be accessed within the class they belong to, thus there is no way to test them from the test class in JUnit. On the other hand, it becomes necessary to directly test the operations of the private method in case of following:

- When a private method contains an algorithm which requires more unit testing than it is possible through the public interfaces.
- When to enhance modularity, developers create private utility methods which do not act on the instance data but simply work upon the passed arguments to produce a desired result.
- When the level of abstraction furnished by public methods of a class could be too high such that the algorithm of private method could not be easily targeted.
- If an error caused by a private method is identified while testing a public method, at times it might take really long to find out the exact location of the error.

One such example of a private method is shown in Listing 6.5. In this example, the *showURLInfo* method of the *ProcessURL* class calls the private *replaceSpaces* method which replaces the unwanted spaces from the input string with *%20* before creating an object of the Java *URL* class from it. We can test the public *showURLInfo* method of this class with JUnit (or other conventional technologies) but as the *replaceSpaces* method is private, it cannot be tested. However, as we can see that the private method in this example implements the intricate algorithm of replacing the spaces with *%20*, it becomes important to directly test it as well. For example, test cases like string containing multiple spaces at the end or two (or more) consecutive spaces in middle, should be checked directly upon the *replaceSpaces* method.

Listing 6.5: A private method with an algorithm

```
public class ProcessURL {
        public void showURLInfo(String str)
        {
                String urlString = replaceSpaces(str);
                try {
                        URL url = new URL(urlString);
                        System.out.println(url.getProtocol());
                        System.out.println(url.getHost());
                        System.out.println(url.getDefaultPort());
                } catch (MalformedURLException e) {
                        e.printStackTrace();
                }
        }
        private String replaceSpaces(String str)
        {
                int noOfSpaces = 0, i;
                for(i = 0; i < str.length(); i++)
                        if(str.charAt(i) == ' ')
                                noOfSpaces++;
                while (str.charAt(i-1) == ' ')
                {
                        noOfSpaces--;
                        i--;
                }
                int index = ((i-1) + noOfSpaces * 2 + 1) - 1;
                char[] charArray = new char[index+1];
                for (int j=i-1; j>=0; j--)
                {
                        if (str.charAt(j) == ' ')
                        {
                                charArray[index] = '0';
                                charArray[index-1] = '2';
                                charArray[index-2] = '%';
                                index = index - 3;
                        }
                        else
                        {
                                charArray[index] = str.charAt(j);
                                index--;
                        }
                }
                return String.valueOf(charArray);
        }
}
```

In order to assist unit testing of private components in JUnit, the Java Reflection API can be used as a fill in. The *java.lang* and *java.lang.reflect* packages provide necessary classes for Java reflection. However, there are several disadvantages of this approach. Firstly, using Reflection API is not easy [81]. The test code becomes verbose, harder to understand and maintain when the reflection API is deployed. Apart from this, since java reflection involves the types that are dynamically resolved at run time, the associated operations have slower performance as certain Java virtual machine optimisations can not be exercised [82]. For all these reasons, using reflection mechanism is not recommended by most of the software developers [83].

However when testing the Java applications using the AspectJ approach, the private members can be easily accessed within the testing aspect by adding *privileged* keyword to the aspect. Code inside privileged aspects has access to all members of the captured object, even the private ones. For example, we used a privileged aspect to test the Java open source download management tool *JDownloader*. Particularly we tested the *validateForm* method of the *AddLinksDialog* class. In this *AddLinksDialog* class, there is a private member *input* which is an object of *ExtTextArea*. We were able to access this private data member using our privileged aspect and provide different values for web links for the purpose of testing the functionality of the *validateForm* method of the *AddLinksDialog* class as shown in Listing 6.6.

Listing 6.6: Testing private members using *privileged* aspect

```
public privileged aspect testJDownloaderAddLinksDialog
{
        pointcut Test4(AddLinksDialog t) : execution(protected void
            ↪ org.jdownloader.gui.views.linkgrabber.addlinksdialog.
            ↪ AddLinksDialog.validateForm()) && target(t);
        void around(AddLinksDialog t) : Test4(t)
        {
                String [] testinput = new String[] { "www.mnit.ac.in",
                    ↪ "www.google.co.in", "y.to", "i.tv", ".", "***", null,
                    ↪ "123", ..... };
                for (int i = 0; i < testinput.length; i++)
                {
                        t.input.setText(testinput[i]);
                        proceed(t);
                }
        }
}
```

Furthermore, the execution time taken for accessing private method of a class using Java reflection in case of JUnit is higher when compared to that of accessing it using a privileged aspect in AspectJ [84]. This difference becomes significant when the number of times a private method is accessed is high. The classes (JUnit) and aspects (AspectJ) meant for testing a method are executed several numbers of times while performing testing. This is either because of multiple test cases for the method or because the test cases have to be re-run every time after bringing about changes in the method under test. Thus if there is a private method under test, it shall be accessed several times while performing testing and hence the total test execution time with testing aspects will be considerably less as compared to that with JUnit test class using reflection mechanism. The bar graph shown in Figure 6.3 depicts a comparison of the execution time taken when a private method with extensive activities (updating considerable number of tuples in a database) is accessed several times using privileged aspect as well as reflection mechanism. It is apparent from this bar graph that when the number of accesses to the private method increased, the gain in execution time using privileged aspect also increased from 11% to 22%.

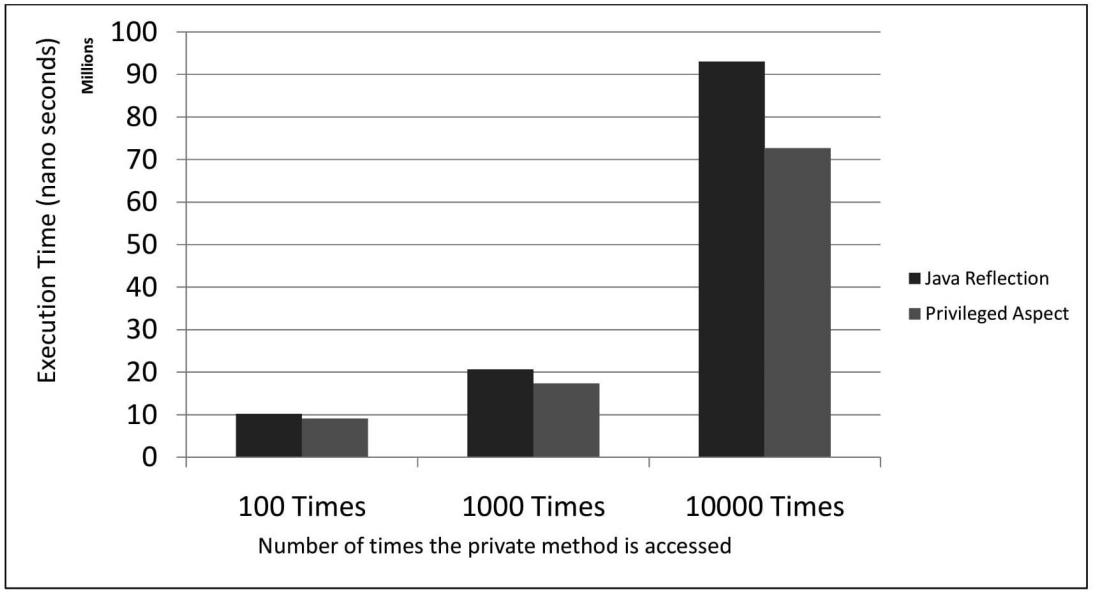

Figure 6.3: Execution time taken for accessing a private method using privileged aspect and Java reflection mechanism, performed on a system with Windows XP SP3 having Intel T6670 Processor and 4GB RAM

By the definition of a unit testing, it is apparent that a unit test suite should test every unit of code which should be irrespective of its scope. Since AOP has provisions for accessing the public as well as private components of the class within the testing aspects, therefore it is a better choice to perform thorough unit testing.

6.2.4 Performing Integration Testing

While testing Java applications, JUnit performs well for conducting tests at unit level only but using AspectJ we can carry out testing at other levels too. Integration testing, which is conducted simultaneously with the development process when all the necessary modules may not be actually available for test, can be carried out using AspectJ. Use of AspectJ simplifies the integration testing phase and it can be applied for integration testing of all types of Java applications. Using aspects in AspectJ, we could create stub or driver in lieu of an application module which is either not fully developed yet or needs extensive resources for execution as discussed in Section 3.4.2 of Chapter 3. Such a stub or driver is useful for performing Integration Testing. AspectJ provides us with around advice which we used to write stubs that completely bypass the execution of the captured joinpoint. Around advice was utilised to write the functionality of a missing module that has to be integrated or to implement a light-weighted alternative of a module. Likewise, we also created drivers using aspects which were used to call a particular API or module under test with the required arguments. Moreover, aspects can also be utilised to mock a known state of database while performing integration testing.

With JUnit there is no such provision for writing stub or driver in lieu of an application module. Thus integration testing is not well supported by JUnit [55]. Other available integration testing tools are meant for specific domain only like embedded systems (VectorCAST), critical software (LDRA) or message based applications (Citrus) but our proposed technique can be used for the integration testing of all type of applications. Integration testing involves setting up the whole environment which resembles the situation in which the system is finally deployed and around advice written within aspects is very useful for this purpose.

6.2.5 Performing Invariant Testing

Invariant testing involves examining conditions that are expected to hold true for a program component or may be for the whole program implementation. Invariants conditions could be run time like a newly created object should not be null or the return value of a method should be within a specified range or a particular method should not be called from any class other than the allowed class. The context at a joinpoint can be captured with the help of constructs like *args*, *after returning advice* or *within* in AspectJ and such context can be used to ascertain the imposed

run time conditions. Static invariant conditions that are required to be tested could be asserting that a particular API is never called or verifying that the private members are set within the setter functions only. Compile-time declarations in AspectJ like *declare warning* or *declare error* can be used to ascertain such static conditions.

Most of the times, the invariant condition is required to be tested across the whole program itself and such testing is thus crosscutting in nature. For example, if we want to test that the value of a program variable should always be within certain constraints (like y=ax+b or a≤x≤b or x∈y) then it would be required to write testing code at all the places within the program code where the program variable is being (directly or indirectly) assigned a value. Using aspects in AspectJ, we can capture all accesses to the variable using the *set* pointcut and further write an after advice to check that the value assigned is within desired constraints. Although, *assert* in JUnit can be used to check for internal invariants within the test class, but there exists no provision for testing the crosscutting invariant conditions.

Daikon tool for the dynamic detection of invariants uses *instrumenters* which instrument the source code and produces a new version of the source code to check for the invariants [85]. With our technique, source code is instrumented with aspects externally. Further, Daikon can test for invariants in C, C++, Java and Perl programs but this framework can be used only for primitive and string types and is not suitable for complex types. *Dynamic Invariant Detection ∪ Checking Engine (DIDUCE)* is another invariant detection tool which is meant for Java programs. Alike AOP, it instruments a program's bytecode. DIDUCE dynamically formulates hypotheses of invariants obeyed by the program and considers an anomaly only when there is a large deviation from the past value for a variable at a program point [86]. Thus, it can be used to provide dynamic invariant violations when a software error occurs but not for testing invariant conditions like "the values of a variable should lie within desired constraints".

6.2.6 Testing for Memory Leaks

Memory leakage testing is important because even a small memory leak in an application can cause the complete available memory to get exhausted over time when the application runs continuously. Garbage Collection (GC) is a mechanism which automatically reclaims memory occupied by objects which are no longer referenced. With languages like C++ which were designed for manual memory

management and lacks a GC, the probability of a left behind memory leak due to the developer's mistake is high. Further even with languages like Java which do have a GC mechanism of their own, memory leaks which are not garbage collected can still prevail as we exemplified in Section 3.1.1 of Chapter 3.

We used aspects to find out if there exists memory leaks due to unreferenced objects when the programmer forgets to call the self written methods for object destruction like *finalize/dispose* or when an inner class object is still alive even after its use within the outer class method is over or when caused by buffer overflow. We could find a memory leak in the *ChartPanel* class of JFreeChart, which is a widely used open source application for preparing bars, charts, histograms etc., with the help of aspects.

Null pointer exceptions can also be detected using our AOP approach. Null pointer exceptions occur when we use a reference that points to no location in the memory. Aspects when used for null pointer detection and handling reduce the number of lines of code, has got better tolerance for changes in the specifications and also avoids undesirable code tangling [45, 87].

There is no reliable way of finding out memory leaks using JUnit. Tools like *VisualVM* [88] and *HPROF* [89] are available for Java to detect memory leaks in applications by monitoring the Java Heap. However during our work, we noticed that although these memory analysis tools can detect the presence of an increasing memory usage in the system, but these do not provide a mechanism for determining the exact location in the source code where the bug causing memory leak exists. Moreover, a constantly increasing memory usage is not necessarily the evidence of a memory leak. Applications use memory to store frequently requested information in the form of cache and if there exists certain design error, such information storage may consume more memory over time but it is not a memory leak. Also, different tools for detecting memory leaks create different memory profiles and if two profilers disagree, they both cannot be correct [90]. Using aspects we could not only check for the existence of memory leaks but also we were able to find out which objects have not been dereferenced.

6.2.7 Performing Servlet Testing

To properly test a servlet we would either have to run it inside a real servlet container or create a mock servlet container. JUnit alone does not suffice to test a servlet application. During testing of a servlet, the actual request and

response are not available, since the servlet container is not running. Therefore we need to mock both the *HttpServletRequest* and *HttpServletResponse* objects to simulate as real and get the desired behavior. For this, we need to use APIs like *Mockito* or *org.springframework.mock.web* to mock out servlet request or response objects. However, before using these APIs we need to add their jar files to the project which increases performance overheads. Considerable learning curve for the testers is also associated with these APIs. An example of JUnit testing code using the *org.springframework.mock.web* API that can be used for the purpose of servlet testing is shown in Listing 6.7.

Listing 6.7: Servlet Testing using JUnit

```
import static org.junit.Assert.*;
import java.io.IOException;
import javax.servlet.ServletException;
import org.junit.Before;
import org.junit.Test;
import org.springframework.mock.web.MockHttpServletRequest;
import org.springframework.mock.web.MockHttpServletResponse;

public class JUnitTest
{
        private MyServlet servlet;
        private MockHttpServletRequest request;
        private MockHttpServletResponse response;

        @Before
        public void setUp() {
                servlet = new MyServlet();
                request = new MockHttpServletRequest();
                response = new MockHttpServletResponse();
        }

        @Test
        public void correctUsernameInRequest() throws ServletException,
            ↪ IOException {
                request.addParameter("username", "scott");
                request.addParameter("password", "tiger");
                servlet.doPost(request, response);
                assertEquals("text/html", response.getContentType());
        }
}
```

On the other hand, when we test servlets using our AOP approach, no separate API is required. We simply make use of the javax.servlet package which is a

part of the Java Enterprise Edition. We used aspects that implements the *Filter* class to override methods like *getparameter* so that the requests from client can be simulated to pass the test data to the servlet under test. Moreover, servlet testing aspects can be automatically created by using our TAGL as we illustrated in Section 5.4.6 of Chapter 5. As apparent from the provided TAGL examples, servlet testing using TAGL is straightforward, does not involve the use of any external API and no additional learning is involved.

6.2.8 Performing Load Testing

JUnit cannot be used for load testing as there is no provision for emulating the workload for system under test. *JMeter* [91] is the most popular open source load testing desktop application which can be used to create multiple users in the form of threads that shall send HTTP (or other) requests to the system under test and evaluate the server's performance by calculating its response time. Although JMeter is most suitable for creating load tests for web applications, databases and FTP servers, it cannot be used for load testing of Java desktop/standalone applications. Likewise *Load Runner* [92] too can emulate concurrent users but for web and database servers only and further a prohibitive cost is associated with it.

However, it is equally important to test the performance of the standalone desktop applications and modules under load when they are critical. For example, when a module is being executed several times or when a module is processing a lot of records, then carrying out load testing becomes condemnatory. Using AspectJ, we could carry out such unit performance tests too as explained in Section 3.3.1 of Chapter 3. Further as illustrated in Section 4.1 of Chapter 4, we load tested *NetC* by creating multiple sockets and determined the application's limits for handling concurrent users using AspectJ; the same is not possible with JMeter or Load Runner.

Using suitable context collecting constructs, information regarding the performance of the system under load test like memory consumption and execution time can also be gathered which can be further utilised for performance tuning. We can identify the slow parts in the application and also compare two different implementations. One of the important challenges for load testing viz. setting up the test environment, can also be addressed by writing appropriate around advice(s) within aspects.

Moreover, for load testing Java based (J2EE) web applications, the first and foremost thing would be to load test the modules individually and this is possible with our approach. Modules which are taking high execution times and thus need to be optimised can be identified by load testing with aspects. When web applications are based on servlets, these can be load tested and the time taken for processing HTTP requests can be measured using aspects, in a way similar to that explained in Section 3.3.2 of Chapter 3.

6.2.9 Testing of Concurrent Applications

With the emergence of multi-core and multi-processor architecture, the development of multi-threaded applications has increased. In these multi-threaded applications, there are multiple threads which execute independently and concurrently so that the available hardware can be used effectively. Different threads are scheduled on different processor cores as shown in Figure 6.4 which saves the execution time. However, the behavior of threads can be confusing and counter-intuitive [93] which leads to non-deterministic nature and unexpected interleavings in the parallel multi-threaded applications. Regardless of the presence of these unpredictable interleavings, a concurrent Java application must work flawlessly and this can only be ensured by thorough testing of the thread synchronization mechanisms used by the developer. JUnit has got no provision for testing concurrent Java applications.

Java does not prevent concurrent bugs like deadlocks or race conditions and the developer has to ensure their avoidance through available synchronization mechanisms. Conventionally, testing for the presence (or absence) of such concurrent bugs involves inserting noise [13] or calls to a suitable scheduling function [46] at concurrent events in the Java program. Such noise insertion or calls to a scheduling function requires the source code to be instrumented at various places and leads to scattered testing code when automation is not deployed for testing. The most widely used automated concurrency testing tool is *ConTest* which was developed by the IBM Haifa Research Laboratory [94]. It instruments at the bytecode level and injects calls to the "ConTest runtime functions" at selected places which try to cause a thread switch or a delay.

The main functions of ConTest are:

- Cause synchronization problems to surface-up more likely while testing

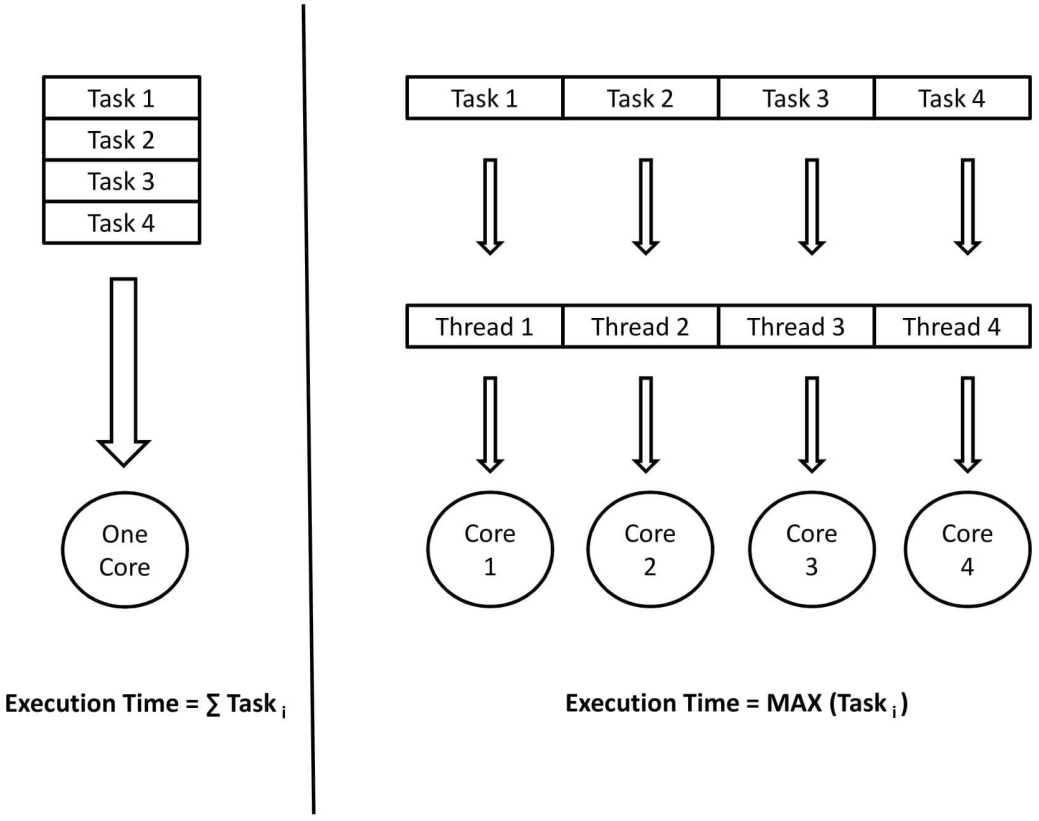

Figure 6.4: Reduced execution times with multi-threaded programming on multiple core CPUs

- Increasing likelihood that a program scenario which gave rise to a specific synchronization problem will recur
- Produces a lot of useful debugging information
- Provides users with thread coverage information

Aspects can be used to externally instrument the source code for increasing the likelihood of catching concurrent bugs [48] without making any changes to the source code. Aspects work by instrumenting the multi-threaded program with conditional sleep and then observe different scheduling scenarios. For example, we used aspects to insert noise after a thread execution, or modify an existing sleep time fixed by the programmer or to insert noise before all accesses to a shared variable as detailed in Chapter 4. The *org.aspectj.weaver.showWeaveInfo* is a system property available in AspectJ that emits a message each time a joinpoint is woven. It can be used to find out which all threads and in what order were covered by the testing aspect.

AOP proves to be useful for noise insertion and addresses the question about "how

to introduce noise" for concurrency testing without transforming the source program under test but it does not throw any light onto "where in the source program should such noise be inserted". We recommend insertion of noise at all program points which are responsible for thread synchronization (like call to *wait*, *notify*, *start* etc.) and where concurrent events occur. Moreover, we inserted random noise at concurrent events using aspects. Even the ConTest default configuration, when its "test random parameter functionality" is enabled, selects noise parameters at random before each execution. But in order to discover concurrent errors effectively, heuristics for insertion of noise should be used. Notwithstanding, our proposed approach is compatible and can be deployed with approaches that implement effective heuristics.

In a nutshell, instrumentation of the application's code for concurrency testing with AspectJ is non-invasive and quite straightforward which is furthermore simplified when TAGL is used for writing the testing code. On the other hand, deploying ConTest requires a lot of expertise.

6.2.10 Context Collection for the Purpose of Debugging

The ultimate goal of software testing is the discovery of bugs. Now-a-days large scale and complex software lead to increasing number of bugs. Once a bug has been identified, the different phases of bug resolution start. One of the most important phases in bug resolution is bug understanding. For a given bug, the assigned developer needs to find the source code files where the bug is and then update the code as a part of the bug-fixing process [19]. Thus if context corresponding to the found bug can be collected at the time of software testing, it can make the process of bug resolution less tedious.

One big advantage of using AspectJ for testing over JUnit and the other conventional testing techniques is that we can provide context information regarding the location in the program where the bug occurs. AspectJ provides us with context collecting constructs like *thisJoinPoint.getThis()*, *thisJoinPoint.getTarget()*, *thisJoinPoint.getArgs()*, *joinPointStaticPart.getKind()*, *joinPointStaticPart.getSourceLocation()*, *joinPointStaticPart.getSignature()* etc. Information gathered by these constructs is helpful for debugging the application. The relevant context collected by the various available constructs in AspectJ is enlisted in Table 6.1.

We used these constructs within the testing aspects such that whenever a bug

Table 6.1: Collecting context useful for debugging

Construct	**Significance**
thisJoinPoint.getThis()	Provides the context information of the currently executing object
thisJoinPoint.getTarget()	Provides the context information regarding the target object
thisJoinPoint.getArgs()	When we use it within an aspect that tests a method or constructor, it returns an array of the arguments where each element refers to each argument and exists in the order in which it appears in the method or constructor
joinPointStaticPart.getKind()	Returns the context information regarding the kind of joinpoint such as method-call or field-set etc.
joinPointStaticPart.getSourceLocation()	Provides useful context information regarding the line number in the source code which corresponds to the discovered bug
joinPointStaticPart.getSignature()	Provides context information regarding the signature of the method or constructor where the bug is discovered by the testing aspect

occurs we are able to get the relevant information regarding the execution point where the bug occurred like the line number in source code, the signature of the surrounding method or class etc. For example, in the Listing 3.6 of Chapter 3, we used thisJoinPoint.getSignature() and thisJoinPoint.getSourceLocation() to report the signature of the method and the source code line number at which null value was returned.

In Listing 6.8, we have used the available constructs in AspectJ for the collection of context. We have taken example of an aspect that instruments the push method of a stack data structure and generates an error along with the necessary context information whenever the stack top reaches full. The output of the listing is shown beneath it.

Listing 6.8: Collecting context useful for debugging

```
pointcut isFull(Stack st) : call(public void Stack.push(long)) &&
    ↪ target(st);
before (Stack st): isFull (st)
{
        if(st.top == st.maxSize-1)
        {
                System.out.println("Stack is full!");

                String targ = thisJoinPoint.getTarget().getClass().
                    ↪ toString();
                String kind = thisJoinPointStaticPart.getKind().toString();
                String sig = thisJoinPointStaticPart.getSignature().
                    ↪ toString();
                String loc = thisJoinPointStaticPart.
                    ↪ getSourceLocation().toString();
                Object arg[] = thisJoinPoint.getArgs();

                System.out.println(targ);
                System.out.println(kind);
                System.out.println(sig);
                System.out.println(loc);
                System.out.println(arg[0].getClass());

                return;
        }
}
```

Output of Listing 6.8:

```
Stack is full!
class Stack
method-call
void Stack.push(long)
stackUserClass.java:9
class java.lang.Long
```

The constructs used and the context collected in Listing 6.8 are discussed hereunder:

- Using thisJoinPoint.getTarget().getClass(), we were able to find the class of the target object. In our example, the target object is of class *Stack* which called the push method that caused the bug.
- Using thisJoinPointStaticPart.getKind(), we were able to find out what kind of program point was it when the bug occurred. In our case, it was a *method-call*.
- The thisJoinPointStaticPart.getSignature() gives the signature of the method which was called when bug occurred. In this example, it was the push method of the *Stack* class, namely *void Stack.push(long)*. The fully qualified method name is provided.
- The thisJoinPointStaticPart.getSourceLocation() construct gives the location in the source code from where the bug was introduced. At line number 9 in the *stackUserClass*, the push method of *Stack* class was called which made the stack top cross the maximum allowed value limit. *stackUserClass* is the class which using the *Stack* class.
- The thisJoinPoint.getArgs() construct was used to capture the arguments that were used to call the push method of the *Stack* class. The arguments class was obtained using getClass() which printed the fully qualified class name of the argument as *java.lang.Long*.

6.3 Summary

This chapter presented the qualitative benefits of using our approach over the conventional testing techniques. Reduced learning curve, testing without source code modification, testing private members efficiently, collection of context at the error location and being able to perform various types of testing are the main advantages of using our approach. A summary of the key advantages of using our approach over the most popular Java testing tool JUnit is given in Table 6.2.

Table 6.2: JUnit vs Our approach: Qualitative Comparison

Testing feature	**JUnit**	**Our proposed approach**
Testing with multiple input values	Not straight forward, use of *Parameterized* class required which is difficult to learn	Simple aspects using *array* and *for loop* to provide multiple values are sufficient. Further, using TAGL requires only the values to be specified in simple TAGL statements.
Testing private members	No direct mechanism except using the Reflection mechanism which makes code verbose and performance slow	Privileged aspect have access to all members of the captured object (performance gain up to 22%). Further TAGL requires only the scope to be stated as private and then generates privileged aspect automatically.
Integration testing	Limited support [55]	Aspects can be used to create stubs and drivers.
Testing compile time invariants	No provision	Compile-time declarations like declare warning or declare error can be used to test for static conditions.
Testing run time invariants	Separate testing code for testing different classes whether the invariant condition holds true in each of these	Wild card pointcuts using which multiple execution points can be captured simultaneously along with suitable advice minimise the testing effort.
Test reports	Informative and well-formatted reports but petite complex to understand	Simple test reports with context information which are easy to understand.
Memory leakage	No reliable way	Different possibilities of memory leaks can be detected using aspects [45]. Using TAGL, separate but simple denotations to be written for detecting different types of memory leaks.

Table 6.2 : JUnit vs Our approach: Qualitative Comparison, continued

Testing feature	**JUnit**	**Our proposed approach**
Servlet testing	No direct mechanism available, external APIs like Mockito or org.springframework.mock.web required	No separate APIs are required, javax.servlet package which is a part of the Java Enterprise Edition is used. Using TAGL, only servlet name, parameter name and values to be used for testing need to be specified.
Testing non-functional properties	Not possible	Possible because of its inherent property of capturing cross cutting concerns into aspects. TAGL makes it further easy for testers to write testing code for testing non-functional requirements.
Error messages for incorrect testing code	Better and meaningful error messages	AOP provides understandable error messages but TAGL error messages are not very informative. Nevertheless, TAGL's simplicity reduces the chances of errors.

Chapter 7

Comparison with Conventional Technologies:Quantitative Analysis

In this chapter, we shall analyse our proposed approach quantitatively by comparing it with the existing testing methodologies. For this purpose, we shall perform a detailed comparison on the basis of lines of testing code, code coverage and test execution times which are important criterion for the assessment of an automated testing methodology. Suitable illustrations of testing scripts have been provided from our approach as well as the conventional testing techniques to perform various types of software testing and establish the benefits of using AOP for testing. Substantially, JUnit and AspectJ have been used in the discussion.

7.1 Lines of Testing Code

The number of lines of testing code affects the time taken by the testers for writing the code. Higher the number of lines required to be written for test scripts, more would be the time spent by the tester. On the contrary, the situation that software projects usually face is that there is more of untested code and less time for writing the test code. At the same time, more lines of test code would mean more efforts for its maintenance. Maintainability is an important aspect of the testing code. A difficult-to-maintain test code is likely to be abandoned. Test code can be made maintainable by keeping test methods short, striving for fewer lines of test code and limiting the test actions to one or two lines of test code [95]. Thus, by adopting a testing technique that reduces the number of lines of testing code would in turn release the test engineers for more demanding and rewarding work.

Using conventional testing techniques (like JUnit, Visual Studio, NUnit test framework etc.), the ratio of number of lines of test code to the number of lines of source code may go as high as 1:1 [96, 97]. Few open source Java applications for which the JUnit test suites have been provided with the source code are listed in Table 7.1 with their test code to source code ratio.

Table 7.1: Lines of test code : lines of source code

Application Name	**Lines of Code (Tested)**	**Lines of Test Code**	**Ratio**
ANT	17608	8121	46.12%
POI	58754	41610	70.82%
LUCENE	22098	21997	99.54%
JODA	17678	46702	264.17%

During our research, we observed that using aspects in AOP for writing the test cases, the number of lines in the testing code is reduced considerably. For example, when we tested a simple average function in a *Student* class in Java, which takes three variables and calculates their average, with multiple input values for the three variables, the testing program could be written with only 18 lines with AspectJ (see Listing 7.1) whereas the same required 31 lines of code when written using JUnit (see Listing 7.2). In fact, JUnit does not provide any direct mechanism for testing a method with multiple input values, rather we have to use the *Parameterized Class*. Parameterized is a runner inside JUnit that will run the same test case with different set of inputs. The JUnit code written for testing a method with multiple inputs using Parameterized class is not straight forward and is difficult to learn and understand whereas the corresponding aspect code is quite straightforward as evident from Listing 7.1 and 7.2. Moreover, our TAGL makes it further easier for the tester to write the testing code. Just 8 English-like TAGL statements are required to test the average method with multiple values for the three variables and compare the actual outcome with the expected outcome as shown in Listing 7.3.

Additionally, if there are multiple methods of a class which are to be tested and are written in the same JUnit test class, and if we test using parameterized dataset, then all test methods in the JUnit class shall be tested with the same dataset. In

Listing 7.1: Testing a method in *Student* class with multiple inputs using AspectJ

```
//Testing with multiple test cases using an around advice with a setup
    ↪ function
public aspect revisedTestingAspect {
        Student s = new Student();
        static int i;
        int expectedResult[] = {2,5,8};
        int[] mark1 = {1,4,7};
        int[] mark2 = {2,5,8};
        int[] mark3 = {3,6,9};

        void setup(int i)
        {
                s.setMarks1(mark1[i]);
                s.setMarks2(mark2[i]);
                s.setMarks3(mark3[i]);
        }

        double around(Student st) : execution(public double
            ↪ Student.getAverage()) && this(st)
        {
                for(i=0;i<3;i++)
                {
                        setup(i);
                        double result = proceed(s);
                        if (result!=expectedResult[i])
                                System.out.println("Error at " + i + "th
                                    ↪ input" + " Expected Result: " +
                                    ↪ expectedResult[i] + " Actual Result: "
                                    ↪ + result);
                }
                return proceed(st); //do original processing
        }
}
```

Listing 7.2: Testing a method in *Student* class with multiple inputs using JUnit

```
import java.util.Arrays;
import java.util.Collection;

import org.junit.*;
import org.junit.runners.Parameterized;
import org.junit.runner.RunWith;

import static org.junit.Assert.*;

@RunWith(Parameterized.class)
public class TestCaseMultipleInputs {
        private int marks1;
        private int marks2;
        private int marks3;
        private double expectedResult;
        private Student st;

        @Before
        public void initialize() {
                st = new Student();
        }

        //Each parameter should be placed as an argument here, every time runner triggers,
            ↪ it will pass the arguments for parameters we defined
        public TestCaseMultipleInputs(int marks1, int marks2, int marks3, double
            ↪ expectedResult) {
                this.marks1 = marks1;
                this.marks2 = marks2;
                this.marks3 = marks3;
                this.expectedResult = expectedResult;
        }

        @Parameterized.Parameters
        public static Collection<Object[]> getInputs() {
                return Arrays.asList(new Object[][] {{ 1,2,3,2 },{ 4,5,6,5 },{ 7,8,9,8 }});
        }

        //This test will run 3 times since we have 3 parameters defined
        @Test
        public void testGetAverage() {
                st.setMarks1(marks1);
                st.setMarks2(marks2);
                st.setMarks3(marks3);
                double result = st.getAverage();
                assertEquals(expectedResult, result, 0);
        }
}
```

Listing 7.3: Testing a method in *Student* class with multiple inputs using TAGL

```
////type: blackbox
////aspectname: TestCaseMultipleInputs
////classname: Student
////methodsignature: public double getAverage()
////setup: public void setMarks1(int m), public void setMarks2(int m), public
    ↪ void setMarks3(int m)
////argumentname: (marks1, marks2, marks3)
////values: (1,4,7) ,(2,5,8) ,(3,6,9)
////expected: 2,5,8
```

JUnit, we cannot have different parameterized datasets for different methods in the same test class. On the contrary, with AspectJ's aspects we can straightforwardly use different testing datasets for different methods even if they are all written in the same aspect as we have separate advice codes for each.

Testing of servlets using JUnit requires use of APIs like *Mockito* or *org.springframework.mock.web* to mock out servlet request or response objects. This leads to increased line of testing code. For e.g., when we tested a two parameter servlet with JUnit using the *org.springframework.mock.web* API, it needed 22 lines of code. Although, when we tested the same servlet using an aspect in AspectJ only 16 lines of code were sufficient. And the biggest advantage in terms of lines of testing code comes with TAGL, wherein simply 3 TAGL statements are sufficient to generate the 16 line servlet testing aspect in AspectJ. Similar observations were found during the fault injection testing of JScreenRecorder and the same has been depicted in Figure 7.1.

The number of lines of testing code are also reduced by the use of *wildcard pointcuts* which are available in AspectJ. For example, the simple pointcut *execution(* *(..))* shall capture the execution of any method regardless of the return or parameter types. Thus if we want to test for the condition *whether any of the methods in the whole program returns null*, which can lead to a null pointer exception, this single pointcut would be sufficient. Another example could be to test how the various methods of an application handle null arguments. We can capture all methods using a single pointcut and instrument so as to pass a null argument and examine the behavior of the methods from the advice. Similarly, other wild card pointcuts can be used to capture *joinpoints* that share common characteristics and then can be tested all at once. However, there is no such mechanism in JUnit and hence for testing different methods even with common attributes, separate testing code has to be written for every method which apparently increases the number of lines of

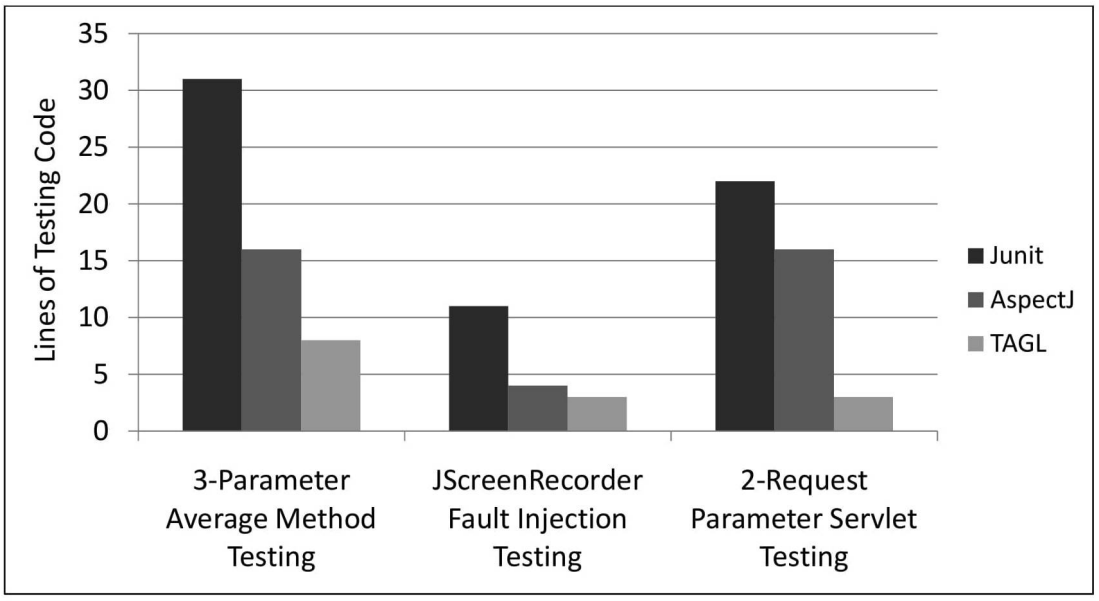

Figure 7.1: Number of lines of testing code is reduced using AspectJ, which further decreases with our TAGL

code.

The software testing phase comprises of about 25-40% time of the total software development life cycle. Research is ongoing in the direction of how the testing efforts can be reduced so that quality software is available in the market in lesser time as considerable amount of time is spent in testing [98]. The number of lines of testing code to be written by the tester is reduced when using AOP for testing which in turn decreases the required testing efforts. The same is further improved by the use of TAGL to a good extent. Thus, TAGL is not only easy to learn but it also relieves the tester from writing lengthy testing codes for testing the production code.

7.2 Test adequacy criteria and code coverage

A *test adequacy criteria* is an assertion regarding which program elements need to be tested so that it can be ascertained that the program has been tested *thoroughly*. If a software passes an *adequate* number of tests, then the developer can be reasonably assured about its correctness or in other words the developer can treat it to be dependable. If a collection of test cases satisfies all the predicates of a test adequacy criteria, it does not mean that such collection performs the complete exhaustive testing of the software or that the software is bug free but it

simply means that such testing activity is *adequate*.

A test adequacy criteria can be thought of as minimum standards that should be followed by a software project which either arise from the specifications of the software or else deemed fit to be tested during the course of development. For example, if a system specification demands that it should be able to handle hardware failure that interrupts data transmission over network, then the test cases for such system should be able to simulate a hardware failure. Likewise if a criteria stated by the developers imposes the obligation that each loop (be it while, for etc.) should be executed once or more to determine initialization related errors, then the test suite should be prepared accordingly. A test adequacy criteria thus provides guidance to the testers in devising a comprehensive test suite. A test adequacy criteria controls the cost of software testing by revealing the missing test cases swiftly and by determining when sufficient testing has been done and can be stopped. In short, the assessment of a test to examine its weaknesses is carried out based on the test adequacy criteria (see Figure 7.2).

A test adequacy criteria which is well qualified for a software system may not be equally suitable for another system. In other words, a criteria might be satisfactory for writing effective test cases for a software system for testing its correctness but a different criteria could be more satisfactory for thorough testing of another system. For e.g. test adequacy criteria have to be framed differently for sequential, concurrent and distributed systems even if they address the same problem space. Only a well framed test adequacy criteria shall lead to test suite that reveals errors in the system or else if the test adequacy criteria is inappropriate, then a test suite that satisfies such criteria might not be able to expose errors in the system. A test adequacy criteria is said to be effective when if a test set that satisfies the criteria passes the program successfully, then the program shall also be passed successfully by another test set that too satisfies this adequacy criteria. However, practically it is not possible to achieve such consistency. "Effective and error-detecting test adequacy criteria" and "how likely a test suite, that satisfies a criteria, detects an error": have been topics of research [99, 100].

We propose that adequacy criteria is actually a crosscutting concern and thus AOP best suits for satisfying it. Aspects in AOP make the selection of the code easy that has to be included in a test adequacy criterion [15]. There are predicates in test adequacy criteria that relates to lexically scattered code across the program under test, thus it is difficult to capture such scattered artifacts by test cases written using conventional testing techniques. Using AOP can relieve the tester

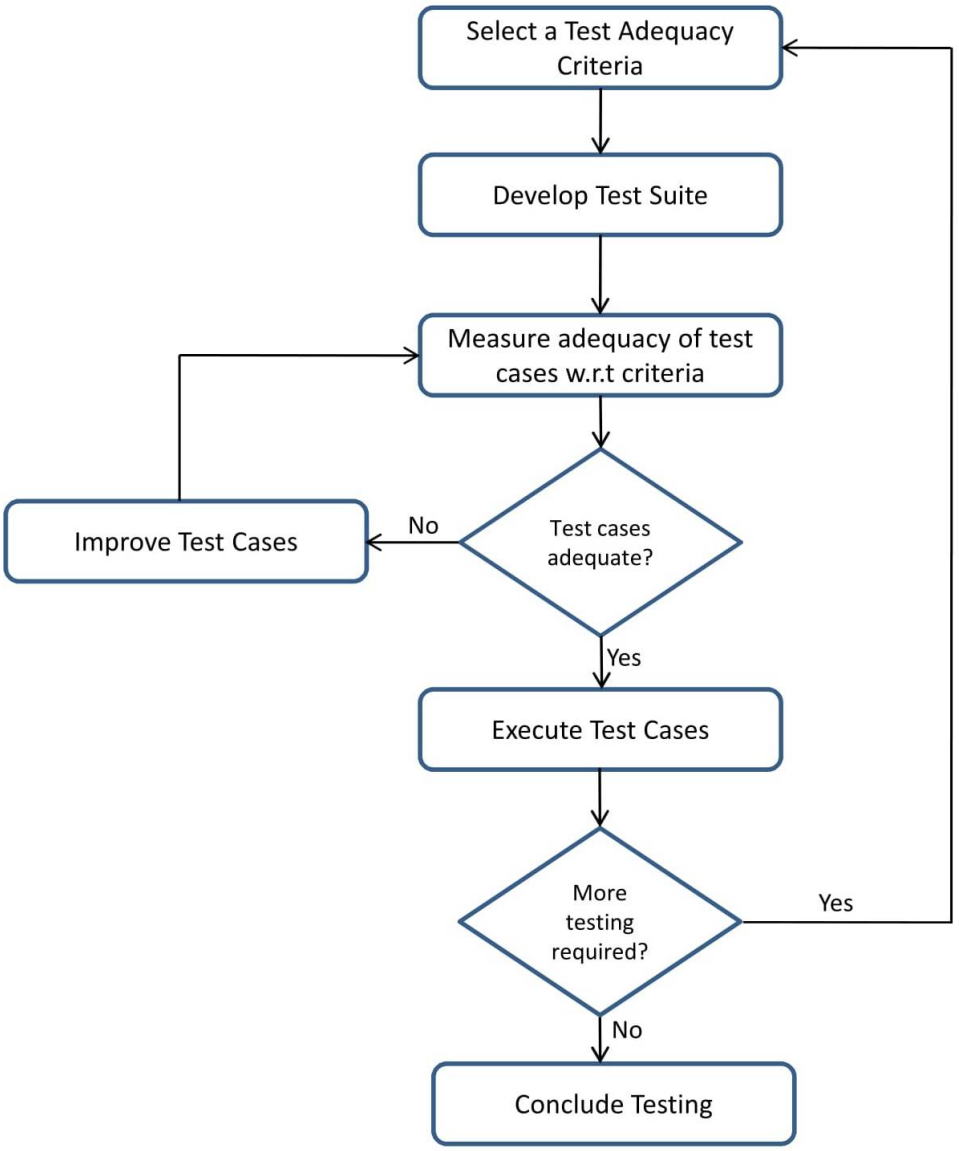

Figure 7.2: Test assessment process

from manually selecting the scattered code specified in the test adequacy criteria.

A test adequacy criteria that asserts predicates related to synchronization, security or robustness shall encompass source code that shall be scattered in various modules. This simply correlates to the fact that these attributes are crosscutting in nature. Using our AOP testing approach, testing aspects can be written using appropriate wild card pointcuts that are adequate to cover the crosscutting source code that spans across various execution points in the code base.

For example, if there is a test adequacy criteria with a predicate that asserts for

coverage of *all the calls that have been made externally to any of the methods of a particular class and all its subclasses* shall involve a lot of tester efforts to be satisfied using the conventional techniques. It is so because there could be enormous calls to such methods which shall be written at several execution points within the code base of program under test. However using AspectJ, this adequacy criteria can be satisfied by single wild card pointcut:

call(* ClassName+.*(..)) && !within(ClassName+)

Here the first part of pointcut *call (* ClassName+.*(..))* shall capture calls to all the methods inside the class with ClassName or its subclasses. This will also match any new method that has been introduced in the subclasses of ClassName. Further the second part restricts the matching to only those calls which are not within the lexical scope of the ClassName class and its subclasses. Further, the testing code to test the covered criteria can be written within the advice.

Another test adequacy criteria that asserts to *cover every instance where a particular field balance of a class Account is being set with a new value* and requires to test that the value should be non-zero, can be captured with a pointcut like this:

set(private float Account.balance) && args(newValue)

Further, an advice can be written which acts before the captured execution points and tests that the *newValue* should be non-zero. With the conventional testing techniques, writing test cases that cover such a test adequacy criteria is not so straightforward.

A test predicate in an adequacy criteria governing synchronization could be to *cover all the calls to the synchronization functions within the source code.* The test predicate further requires to test the system's behaviour and adherence to concurrency when an unavoidable delay emerges. The following pointcut can be used to capture calls to the synchronization methods like *wait*, *notify*, and *notifyAll* present in a concurrent program:

void around() : call(* *.wait()) || call(* *.notify(..)) || call(* *.notifyAll(..))

An appropriate advice with a heuristic noise can then be used to simulate a delay and evaluate whether the system is able to maintain the desired concurrency requirement or not.

Code coverage refers to amount of application code that has been exercised by a set of test cases. It is a way to ensure that the test cases are actually testing the application code. If a test adequacy criteria has to be covered completely, then 100% coverage is required. However in practical testing scenario, it is very difficult to achieve 100% coverage of a test adequacy criteria because of the following reasons:

- There could be some parts of code which are not reachable
- The system under test is not critical enough such that 100% coverage is necessary
- The planned time-frame for testing is limited
- The available automated testing tools are not capable to achieve 100% coverage
- The testers are not skilled enough
- Attempting so can delay time-to-market and slip an early release

Most organisations consider about 85% coverage as sufficient so as to produce a quality software [101].

7.2.1 Types of test adequacy criteria

There are various ways in which the test adequacy criterion is classified. However, irrespective of the type of the criteria, the basic testing principle remains the same and the actual output is compared with the expected output as per the software requirement to identify the bugs.

When classified in accordance with the source of information, there are primarily two categories of test adequacy criteria: *Specification based* and *Program based*. In a specification based test adequacy criteria, the requirements of the software are fully exercised. In case of program based criteria, the predicates are determined based on the features of the program under test.

Further test adequacy criteria can also be classified in accordance with the testing approach that shall be undertaken. Possible approaches for testing could be fault based testing approach, error based testing approach or structural based testing approach. This classification is depicted in Figure 7.3 and discussed hereunder.

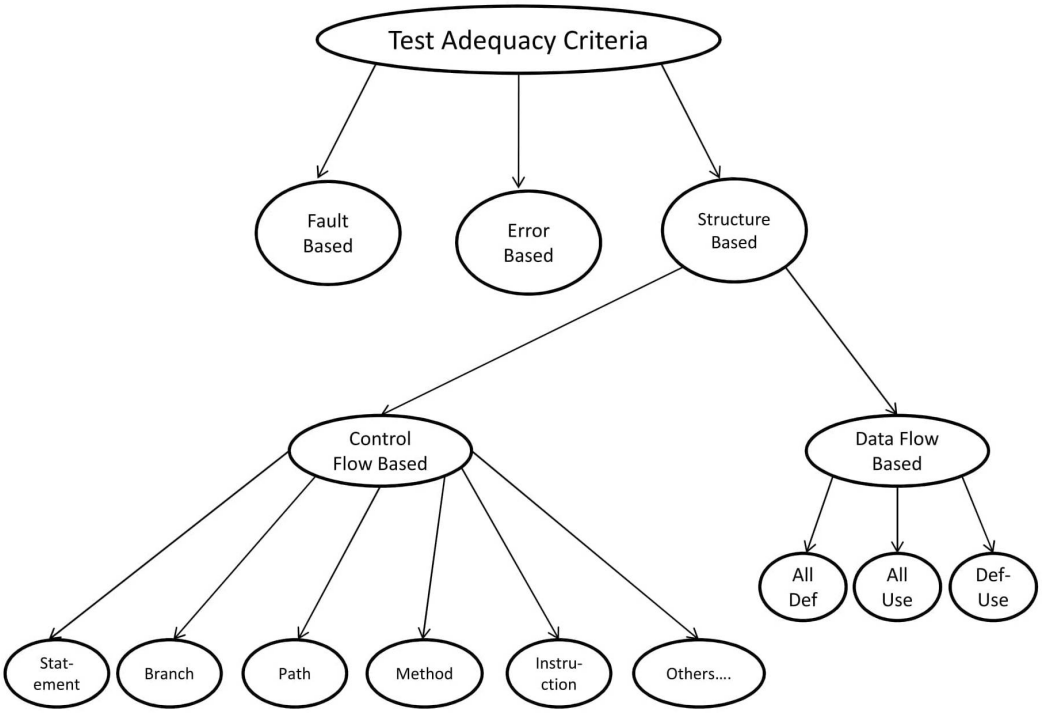

Figure 7.3: Different types of test adequacy criteria

- Fault-based testing: Here the test adequacy criteria sets a measure for the fault detection capability of the test suite
- Error-based testing: In this case, the test adequacy criteria advocates the testing of error-prone points based on the knowledge about the occurrence of common errors in the system
- Structural testing: In case of structural testing, the test adequacy criteria predicates the coverage of particular set of elements in the structure of the program

In case of structural based testing approach, the test adequacy criteria can be further classified into following important types:

- Control-flow based adequacy criteria
    1. Statement coverage: Statement coverage corresponds to the physical coverage of the code. It reflects the percentage of statements that have been executed by the underlying test suite. The statement coverage adequacy criteria is simple and fundamental but it is actually a weak measure. For example, if the defect is a missing statement, it may remain

undetected by tests satisfying complete statement coverage. Another example could be that of an *if* statement without an *else* where the *false* condition will not be exercised even with 100% statement coverage.

2. Path coverage: Path coverage criteria is to ensure that every possible path in a code is executed at least once. However, achieving full path coverage is impractical because for n decisions within a module, there could be 2^n paths.
3. Branch coverage: This criteria enforces the condition that each branch of every control structure should be executed. For e.g. in case of an *if* statement, both the *true* and *false* branches should be covered.

   Branch Coverage = (Number of Branches Covered)/(Total Number of Branches)* 100

4. Instruction coverage: It correlates with the amount of Java instruction bytecode that is covered by the test suite. There could be many logical expressions on one Java statement and thus a single statement may be compiled to multiple bytecode instructions.
5. Method coverage: It is a metric to measure how many methods out of the total methods were entered during the execution of the test suite.
6. Cyclomatic number criterion: This criteria depends on the cyclomatic complexity of the control flow graph of the program under test. It says that the number of test cases that are sufficient for the coverage of a program are equal to its cyclomatic complexity which is calculated as hereunder:

   cc = e - n = 2p

   where cc=cyclomatic complexity of the control flow graph, e=number of edges, n=number of vertices, p=number of connected components

- Data-flow based adequacy criteria: In this case, the predicates assert the execution of certain Definition-Use (Def-Use) associations (pairs consisting of a definition and a use) that exist in the program under test. Data-flow based adequacy criterion is further of the following three types:

  1. All definitions criterion: Each definition to some reachable use
  2. All uses criterion: Definition to each reachable use
  3. All def-use criterion: Each definition to each reachable use

In the next section, we shall compare the code coverage achieved by conventional testing techniques with that achieved by our AOP approach. We shall particularly focus upon Instruction, Branch and Method coverage which are considered to be important coverage criteria [102].

7.2.2 Comparing code coverage for various test adequacy criteria

Aspects provide us with various wild cards pointcuts with which we can cover complex crosscutting code using minimum testing code. Conventional testing techniques (like JUnit) do not provide a clear-cut mechanism for selection of scattered code.

Wildcard pointcut allows us to capture multiple execution points within the source code under test with very few lines of code and thus achieve enhanced code coverage with lesser testing code. For example, the pointcut *withincode(* Library.issueBooks(..))* can be used to capture all the joinpoints inside the lexical scope of *issueBooks* method of the *Libray* class and then test that the limit of number of books that can be issued to a single person is not crossed anywhere. Moreover with the help of wildcard in pointcuts, we can capture the joinpoints even if we do not have full details of the program under test. For e.g., all the public methods of a Banking class that change the state of a variable can be captured using the pointcut *public void Banking.set*(..)*, if the actual names of methods are not known.

Automated tools are widely used by researchers, practitioners and end-users to determine the code coverage metrics for evaluating the quality of tests. We used *EclEmma (version 2.3.3)*, a popular Java code coverage tool, to compare the quantitative measure of the application code that is covered by the test codes written using JUnit and our proposed AspectJ technique. It adds a coverage mode which appears like *Coverage As* similar to the *Run As* option in Eclipse. EclEmma calculates the code coverage by instrumenting the bytecode of the class files with additional code. It highlights the source code with different colors to facilitate the understanding of the tester regarding the coverage. Green highlighting means complete code covered, yellow indicates partial coverage and red signifies the uncovered code. EclEmma generates a variety of coverage reports and charts in Hypertext Markup Language (HTML), eXtensible Markup Language (XML), and

Comma Separated Values (CSV) formats. The coverage view lists the coverage summary for the Java project under test.

As a case study, we used JGAP [103] which is a Genetic Algorithms Package in Java. The number of lines of source code in JGAP is 23,579 and number of lines of test code is 19,340. There are a total of 267 classes and 184 test classes. JGAP is highly test-driven and currently, over 1400 test cases written using JUnit are featured by JGAP. The central idea behind the JGAP package is genetic algorithms which have chromosomes at its heart. Chromosomes constitute a potential solution to the problem under consideration. The chromosome is divided into multiple genes where each gene corresponds to a unique feature of the solution. At every step the system evolves and every chromosome is treated with genetic operators and then tested against a fitness function for selection into the next generation. JGAP actually imitates the natural human evolution process at every step in order to find out the best possible solution.

JGAP does the evolutionary work to find the best possible solution but has got no knowledge regarding the problem under consideration. Thus the developer has to decide and provide the chromosomes along with the genes. The meaning of each gene has also to be rendered to the genetic algorithm. Similarly, the fitness function, which determines the goodness of a potential solution as compared to another potential solutions, has to be provided by the developer. The fitness function should return an integer value against a potential solution that indicates the fitness of the solution, i.e., a higher value means a better solution.

As a part of our case study, we have rewritten tests for more than 100 of the JGAP classes using AspectJ and measured the achieved code coverage with EclEmma. We compared the code coverage using our AspectJ approach to the coverage obtained using the JUnit test classes provided along with the JGAP package. We particularly focused upon three types of coverage, namely Instruction coverage, Branch coverage and Method coverage; these three coverage criteria being important ones [102]. We observed that the coverage achieved with AspectJ testing aspects is better than that obtained with the provided JUnit test classes.

Lets take example of the FileKit class of JGAP package which has got 532 lines of source code. It contains helper functions related to the file system. As the class indicates, FileKit class has got functions to perform operations on files and directories like copy the contents of a file to another, extract file name from a file path, removes unwanted separators from the Uniform Resource Locator (URL) inputs, delete a file from disk, delete a directory from disk, get version of the

module represented by an input jar file, convert an ordinary file name into jar file name, create a directory with given name, reads text from a file with input file name, get files from a directory which match an input pattern etc.

The tests folder which contains all the unit tests for JGAP classes contains a JUnit test class FileKitTest.java under org.jgap.util package. This JUnit test class contains test cases for testing the methods of the FileKit class. The total number of line of testing code of this test class are 129. The coverage achieved for different criteria for the FileKit class when testing it with this JUnit test class as calculated from EclEmma tool is reproduced in Figure 7.4.

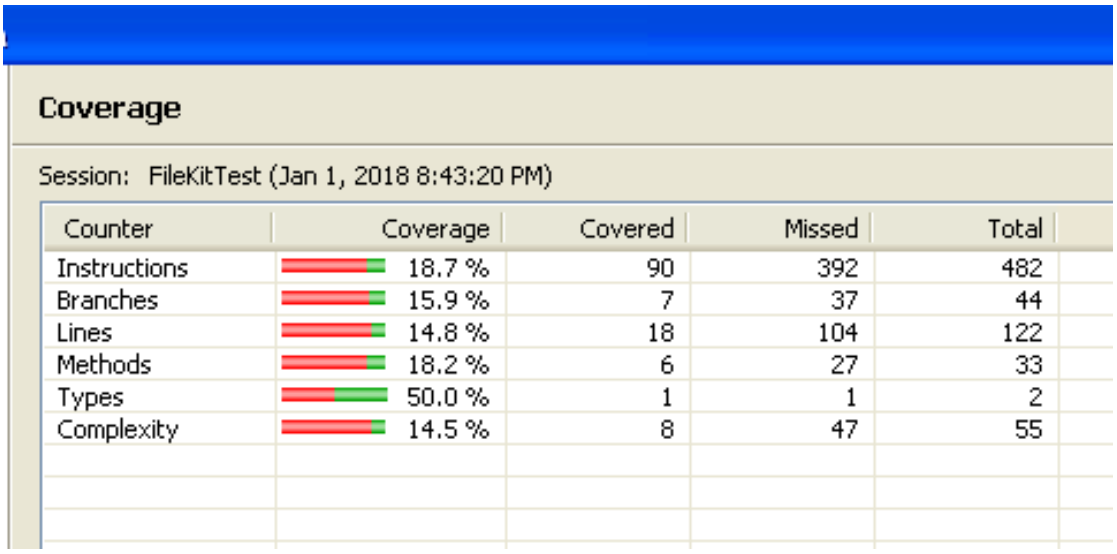

Figure 7.4: Different metrics of source code coverage of FileKit class using JUnit test cases

When we tested the FileKit class using our proposed AspectJ approach, it took only 45 lines of testing code and the achieved code coverage calculated using the EclEmma tool was found to be improved. All coverages namely, instruction, branches, line, method and complexity were improved as evident from the EclEmma snapshot shown in Figure 7.5. A comparison of the code coverage for the important criteria of instruction coverage, branch coverage and method coverage using the JUnit and AspectJ techniques is shown in Figure 7.6.

Likewise improved code coverage was obtained when the other featured test classes were implemented using test aspects. A comparison of instruction coverage obtained using JGAP's featured JUnit tests and tests written in AspectJ, taken with the EclEmma tool for the various classes of JGAP is shown in Figure 7.7. The branch and method coverage were also improved as shown in Figure 7.8 and 7.9. The increased code coverage obtained with Aspect tests is a result of the

Coverage

Session: FileKit (2) (Jan 1, 2018 8:33:47 PM)

Counter	Coverage	Covered	Missed	Total
Instructions	34.2 %	165	317	482
Branches	25.0 %	11	33	44
Lines	27.0 %	33	89	122
Methods	33.3 %	11	22	33
Types	50.0 %	1	1	2
Complexity	23.6 %	13	42	55

Figure 7.5: Different metrics of source code coverage of FileKit class using AspectJ testing aspects

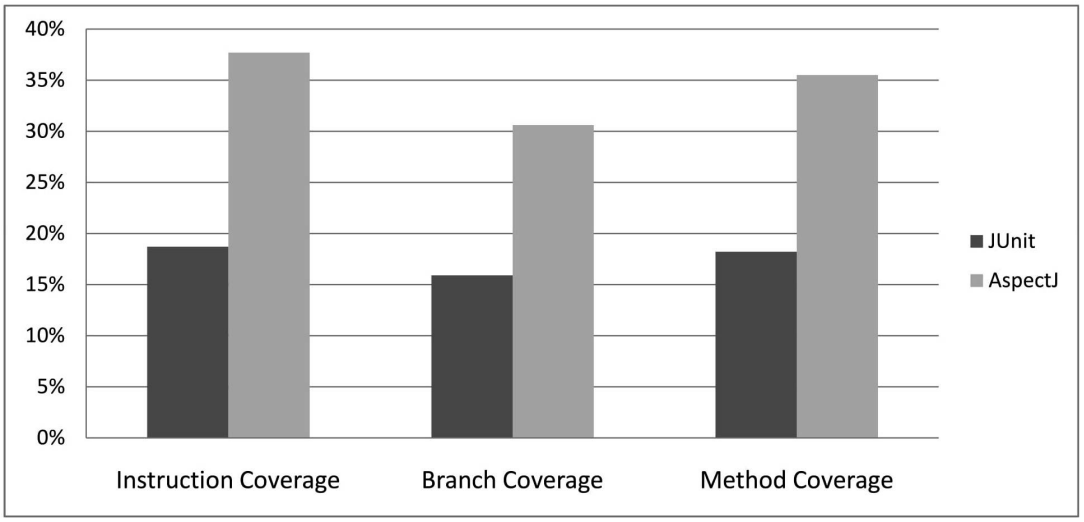

Figure 7.6: Instruction, branch and method coverage as observed using EclEmma for the testing of *FileKit* class. LOC which were 129 in JGAP's featured JUnit test *FileKitTest* were reduced to 45 in our AspectJ testing aspect.

additional testing features available when using AspectJ for testing like memory leakage testing, invariant testing, testing of private members, testing after fault injection, testing the return values of methods with *after returning*, wild card pointcuts etc.

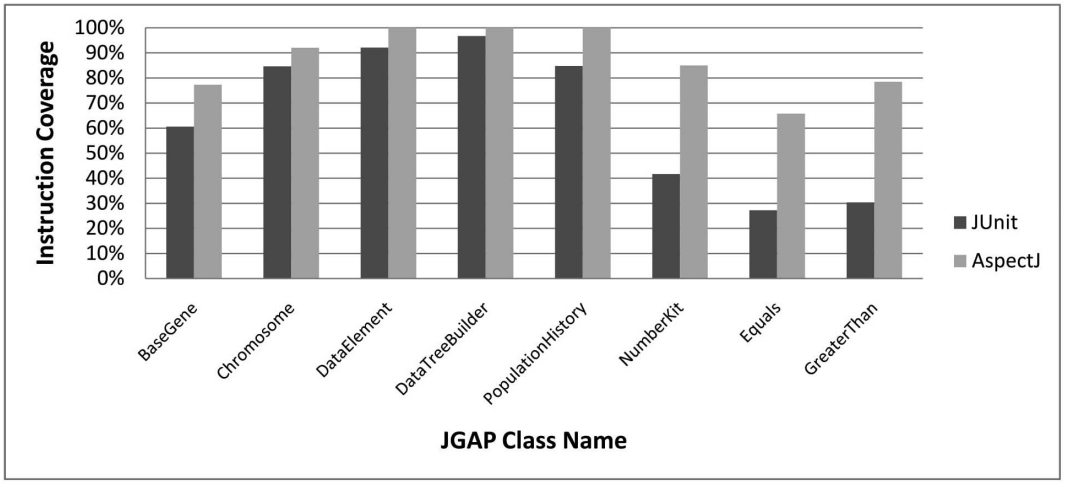

Figure 7.7: Improved instruction coverage for various classes of JGAP with AspectJ testing aspects as compared with the JGAP featured JUnit tests, calculated with EclEmma tool

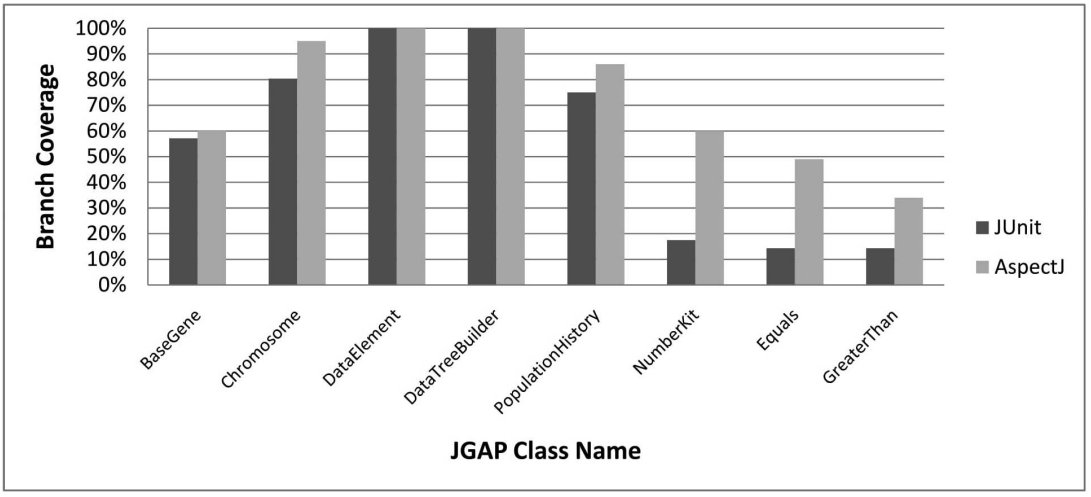

Figure 7.8: Improved branch coverage for various classes of JGAP with AspectJ testing aspects as compared with the JGAP featured JUnit tests, calculated with EclEmma tool

7.3 Test Execution Time

The number of test cases for testing bigger projects is too high [104] and practically it is quite time consuming to test the software with all the test cases using the conventional testing techniques. We observed that the execution time for running test cases is shorter with our proposed AOP testing technique. For example, we tested a simple two argument function using JUnit and AspectJ with multiple test cases for the two inputs. The function returned an integer value which was

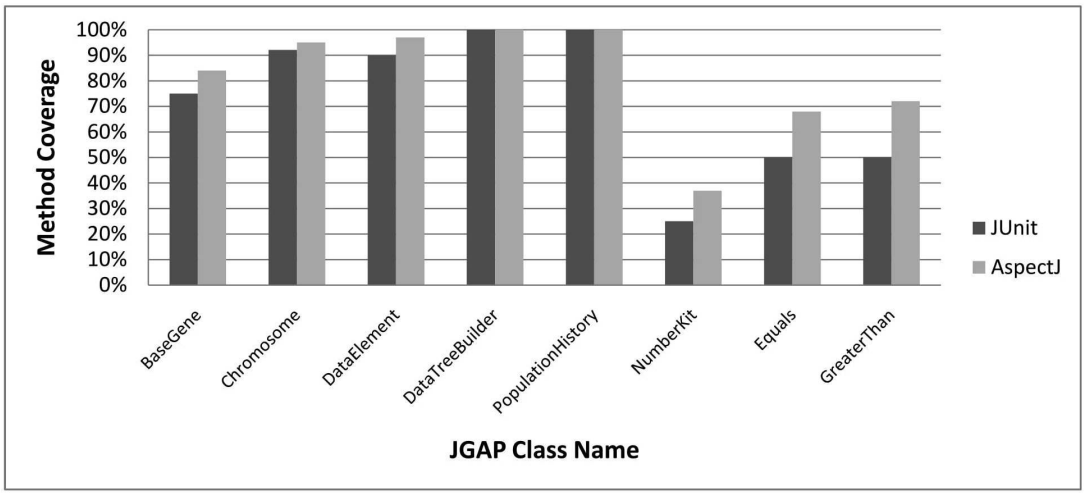

Figure 7.9: Improved method coverage for various classes of JGAP with AspectJ testing aspects as compared with the JGAP featured JUnit tests, calculated with EclEmma tool

compared with the expected output. The observations depicting the test execution times have been shown in Figure 7.10.

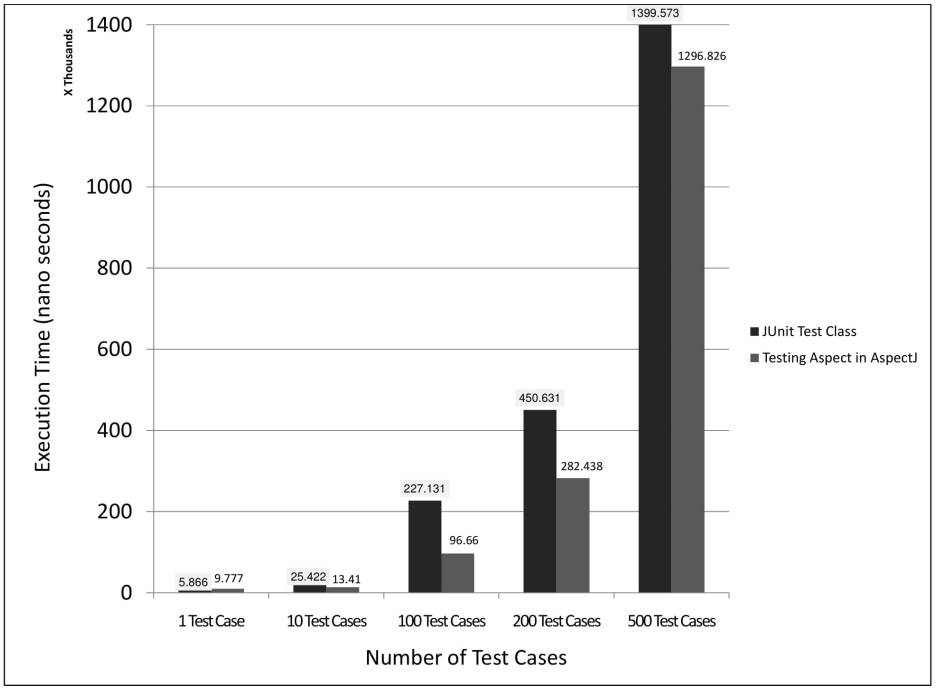

Figure 7.10: Test execution times for testing a 2-argument function using JUnit and AspectJ, performed on a system with Windows XP SP3 having Intel T6670 Processor and 4GB RAM

As shown in Figure 7.10, for a single test case, JUnit took 5866 nanoseconds

whereas Aspect took 9777 nanoseconds for executing the test case. Thus, the time taken using JUnit was shorter as compared to that using AspectJ when only one test case was executed. But for higher number of test cases, which is mostly the case in practical software testing scenario, the execution time taken by AspectJ testing aspect was quite less as compared to the time taken by the JUnit test class as evident from the bar graph in Figure 7.10.

Moreover, the execution time for testing private members using JUnit, which does not have a direct mechanism and thus Java reflection API is used, is higher when compared with that of the privileged aspect in AspectJ used for accessing the private member to be tested. We depicted the same in Figure 6.3 of Chapter 6.

7.4 Summary

In this chapter, we compared the important metrics of lines of testing code, code coverage and execution times using our proposed methodology with that of the conventional testing methodologies. A summary of the results delineated in this chapter is provided in Table 7.2.

Table 7.2: JUnit vs Our approach: Quantitative Comparison

Testing feature	**JUnit**	**AOP approach**	**TAGL approach**
Lines of testing code	More lines	Reduced upto 40% with AOP	Further reduced upto 85% when we use TAGL
Code coverage	Less code coverage	Improved due to possible coverage of private members, memory issues, return values etc.	Alike AOP
Test execution time	Higher execution times for multiple test cases	Reduced upto 60%	Alike AOP

Chapter 8

Conclusion and Future Work

In this Chapter, we summarise the particulars of the AOP and TAGL methodologies presented by us for performing various types of software testing and the key research results thereof. The efficacy and advantages of the contributions of our novel research work are concisely outlined. We also identify and describe the limitations of our proposed approach in this chapter. In the end, we conclude this thesis by specifying the possible future work in the direction of our research.

8.1 Summary and impact of the research

This thesis has proposed the use of Aspect Oriented Programming (AOP) methodology for the purpose of automated software testing and further devised a domain specific language called Testing Aspect Generator Language (TAGL) which provides a high degree of abstraction and automatically generates the testing aspects. A list of common test automation challenges as presented by Fewster and Graham [105] and Antonia Bertolino [106] and how we addressed those with our approach for the automation of test execution and reporting are summarised in Table 8.1.

The first contribution of our research work in the field of software testing is the deployment of AOP as an automated testing technique which not only reduces the testing efforts but also provide for provisions like creating surrounding environment for integration testing, regression testing, analysis of results etc. There are numerous tools arising to support the testing process which can be used in different areas of testing but it is difficult to distinguish which all testing tools should be preferred that can lead to the development of a reliable software project

Table 8.1: Test automation problems

Issue	**How our approach helps?**
Time to market	With AOP the tests can be reused. Test execution time and test code scripting time using TAGL are shortened. This in turn shortens the time to market.
Perform tests which are difficult/impossible to do manually	We can use AOP to perform memory leakage, performance, interference, load testing etc. which are otherwise perverse to perform.
Better use of resources	The repetitive tasks can be automated using AOP which in turn releases test engineers for more demanding and rewarding work.
Running regression tests on a new version of the program	Testing code is localised within the aspect and new test cases in addition to the older test cases can be added to the testing aspect.
Test oracles	To deal with the issue of deciding whether a test outcome is acceptable or not, AOP can be used to write test oracles that validate the test outcome with the expected result.
Education of software testers	AOP languages like AspectJ are easy to learn and moreover, use of the TAGL developed by us does not require deep technical expertise and thus further simplifies the learning curve for testers.
Coherent testing of functional and non-functional properties	AOP can be used for functional as well as non-functional testing (like security, robustness, concurrency etc.) because of its inherent property of capturing cross cutting concerns into aspects.

in accordance with the organisation's intents. However, using solely the AOP approach, most of the important types of software testing can be performed [48, 107]. The use of AOP approach in the field of software testing led to several benefits as discussed in Chapter 6 and Chapter 7. Key advantages of our proposed approach are enlisted hereunder:

- Obliterated test code scattering which surfaces as an issue in testing processes like security testing, invariant testing, memory leakage testing etc.
- Performed concurrency testing by injecting noise in a non-invasive manner that produces interleavings which might cause errors in concurrent applications
- Improved code coverage

- Reduced lines of testing code
- Collected context useful in the debugging process
- Tested private members as well

Secondly, the proposed approach has been implemented and evaluated on a set of widely used open source software programs. We were able to find out remarkable bugs in open source Java projects like JDownloader [59], JFreeChart [61], JScreenRecorder [60], NetC [58], JGAP [103] etc. using our approach and received the acknowledgements for the same.

Last but most important, to overcome the learning issue associated with the automated testing tools, we devised a domain specific language (DSL). Our DSL named *Testing Aspect Generator Language (TAGL)* is useful for the testers of Java applications. It provides a new level of abstraction and decreases the tester's efforts by reducing the learning curve as well as number of lines to be written for the testing code. Using TAGL, software testers who have strong knowledge of the application's business domain but not that of the testing tool, can perform the testing tasks efficiently and thus the bug discovery process is accelerated. Even the testers who are not skilled in AspectJ can still avail the benefits of testing Java applications using AspectJ with the help of our TAGL.

We assert that in general, our proposed approach for AOP based testing is not limited to Java applications. Rather AOP testing approach can be applied to applications written in other languages as well because there exists well developed AOP implementations for many programming languages.

8.2 Limitations and future work

In order to be able to exploit all the benefits of testing using AOP, the software systems have to be designed in such a way that facilitates the separation of cross-cutting concerns. It is so because all the possible execution points in a program written in a language cannot be exposed with the available pointcuts in the corresponding AOP implementation. For example, *for loops* and *array field set* in Java are not exposed and thus cannot be captured using pointcuts in AspectJ [18]. Therefore deploying AOP for effective testing of applications makes it necessary to bring about certain modifications in the development procedure too. Subtle work

has been done in this regard by different researchers to implement the missing pointcuts in AspectJ like Chen et. al [108] have extended AspectJ to expose array joinpoints or like Harbulot et. [109] al presented a model of loop joinpoints in their paper. The outcome of these studies can be consolidated and utilised to develop an extension of AspectJ that addresses all the possible joinpoints. We leave this issue as a scope for future work.

Secondly, although the AOP implementations exists for almost all programming languages but all implementations are not alike or fully developed. AspectJ was the first AOP implementation which has got a very active following in the Java developer's community. It happens to be the fully featured de-facto standard for AOP. AspectC++ has quite similar syntax and semantics as AspectJ (the only difference being that in AspectC++, the program code is changed by a weaver on a pre-processing step before compilation) and furnishes all the features of AspectJ. However AspectR (AOP implementation for Ruby), AspectL (AOP implementation for LISP), AspectMatlab, AOP-PHP, Aspect Python etc. do not fully implement all the AOP functionalities like AspectJ. The difference arises because of the limited constructs which are available and the usability of these constructs. For example, AspectR does not support the around advice and AspectMatlab does not have mechanism for capturing the exception handler execution points. Thus using the AOP implementations of these languages for the purpose of testing might put forward certain limitations. For maximum benefits, it becomes necessary that certain AOP implementations which are so far naive should be evolved with full features.

TAGL is currently limited to generating testing aspects in AspectJ for the testing of Java applications only. It can be extended to produce testing aspects for applications written in other languages as well. Further a GUI (Graphical User Interface) based tool can be evolved for helping the testers. *Drag and drop* feature can be provided to specify the TAGL statements which are then translated into testing aspects. Such a GUI based tool can simplify the selection of program elements that are to be tested and suggestions for candidate test cases can also be provided. Preliminary results obtained in this direction are encouraging, although complete GUI development remains as a piece of future work.

Bibliography

[1] I. Hooda and R. Singh Chhillar, "Software test process, testing types and techniques," *International Journal of Computer Applications*, vol. 111, pp. 10–14, 02 2015.

[2] G. Saini and K. Rai, "An analysis on objectives, importance and types of software testing," *International Journal of Computer Science and Mobile Computing*, vol. 2, no. 9, pp. 18–23, 2013.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *European Conference on Object-Oriented Programming (ECOOP)*, Finland, June 1997.

[4] R. Miles, *AspectJ Cookbook*. O'Reilly Media, Inc., December 2004.

[5] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *IEEE Transactions on Software Engineering*, vol. 1, no. 2, pp. 156–173, 1975.

[6] K. Bareja and A. Singhal, "A review of estimation techniques to reduce testing efforts in software development," in *2015 Fifth International Conference on Advanced Computing & Communication Technologies*, February 2015, pp. 541–546.

[7] M. Hanna, N. El-Haggar, and M. Sami, "Reducing testing effort using automation," *International Journal of Computer Applications*, vol. 81, no. 8, pp. 16–21, 2013.

[8] A. Bamotra and A. K. Randhawa, "Software testing techniques," *International Journal of Innovative Computer Science and Engineering*, vol. 4, no. 3, pp. 122–126, 2017.

[9] K. M. Mustafa, R. E. Al-Qutaish, and M. I. Muhairat, "Classification of software testing tools based on the software testing methods," in *2009 Second*

International Conference on Computer and Electrical Engineering, vol. 1, December 2009, pp. 229–233.

[10] S. Uspenskiy, "A survey and classification of software testing tools," Master's thesis, Lappeenranta University of Technology, Lappeenranta, January 2010.

[11] V. Ribeiro, "Testing non-functional requirements: Lacking of technologies or researching opportunities?" in *Brazilian Symposium on Software Quality*, 10 2016, pp. 226–240.

[12] E. Duclos, S. L. Digabel, Y. G. Gueheneuc, and B. Adams, "Acre: An automated aspect creator for testing C++ applications," in *IEEE 7th European Conference on Software Maintenance and Reengineering*, 2013, pp. 121–130.

[13] Z. Letko, "Analysis and testing of concurrent programs," *Information Sciences and Technologies*, vol. 5, no. 3, pp. 1–7, September 2013.

[14] S. D. Stoller, "Testing concurrent java programs using randomized scheduling," *Electronic Notes in Theoretical Computer Science*, vol. 70, no. 4, pp. 142 – 157, 2002, rV'02, Runtime Verification 2002 (FLoC Satellite Event).

[15] H. Rajan and K. Sullivan, "Generalizing AOP for aspect-oriented testing," in *4th International Conference on Aspect Oriented Software Development*, 2005, pp. 14–18.

[16] P. S. Kochhar, F. Thung, and D. Lo, "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, March 2015, pp. 560–564.

[17] M. Voelter, *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. CreateSpace Independent Publishing Platform, January 2013.

[18] R. Laddad, *AspectJ in Action*. Dreamtech Press, 2005.

[19] T. Zhang, H. Jiang, X. Luo, and A. T. Chan, "A literature review of research in bug resolution: Tasks, challenges and future directions," *The Computer Journal*, vol. 59, no. 5, pp. 741–773, 2016.

[20] G. Kiczales, "Aspect-oriented programming," *ACM Computing Surveys*, vol. 28, no. 4es, Article No. 154, December 1996, doi:10.1145/242224.242420.

[21] O. Spinczyk, *AspectC++ Downloads*, accessed July 1, 2017, https://www.aspectc.org/Download.php.

[22] T. Aslam, J. Doherty, A. Dubrau, and L. Hendren, "AspectMatlab: An aspect-oriented scientific programming language," in *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, ser. AOSD '10. New York, NY, USA: ACM, 2010, pp. 181–192.

[23] I. C. Maries, *aspectlib 1.4.2 : Python Package Index*, accessed July 1, 2017, https://pypi.python.org/pypi/aspectlib/1.4.2.

[24] G. Cro and J. Salleyron, *PECL :: Package :: AOP :: 0.2.2b1*, accessed July 1, 2017, https://pecl.php.net/package/AOP/0.2.2b1.

[25] J. L. Goldman, G. Abraham, and I.-Y. Song, "Generating software requirements specification (ieee-std. 830-1998) document with use cases," in *IRMA International Conference*, May 2007, pp. 552–556.

[26] D. Harekal and V. Suma, "Article: Implication of post production defects in software industries," *IJCA Proceedings on International Conference on Communication, Computing and Information Technology*, vol. ICCCMIT 2014, no. 1, pp. 10–13, March 2015.

[27] B. Beizer, *Software Testing Techniques*, 2nd ed. Itp - Media, 1990.

[28] D. Rafi, K. Moses, K. Petersen, and M. Mantyla, "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey," in *7th International Workshop on Automation of Software Test (AST)*, 2012, pp. 36–42.

[29] J. Srivastaval and T. Dwivedi, "Software testing strategy approach on source code applying conditional coverage method," *International Journal of Software Engineering & Applications (IJSEA)*, vol. 3, no. 3, pp. 25–31, 2015.

[30] S. Shivaprasad and N. Prasad, "Unit testing concurrent Java programs," *International Journal of Computer Applications*, vol. 67, no. 10, pp. 41–46, April 2013.

[31] J. M. Bruel, J. Araujo, A. Moreira, and A. Royer, "Using aspects to develop built-in tests for components," in *The 4th AOSD Modeling with UML Workshop*, San Francisco, USA, September 2003, pp. 1–8.

[32] J. Stamey and B. Saunders, "Unit testing and debugging with aspects," *J. Comput. Sci. Coll.*, vol. 20, no. 5, pp. 47–55, May 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1059888.1059894

[33] L. Yang, "Using AOP techniques as an alternative test strategy," in *32nd Pacific Northwest Software Quality Conference 2014*, Portland, Oregon, September 2014, pp. 107–116.

[34] J. Pesonen, "Extending software integration testing using aspects in Symbian OS," in *IEEE Testing: Academic and Industrial Conference - Practice And Research Techniques*, 2006, pp. 147–151.

[35] J. Metsa, M. Katara, and T. Mikkonen, "Testing non-functional requirements with aspects: An industrial case study," in *Seventh International Conference on Quality Software (QSIC 2007)*, October 2007, pp. 5–14.

[36] X. Li and X. Xie, "Research of software testing based on AOP," in *2009 Third International Symposium on Intelligent Information Technology Application*, vol. 1, November 2009, pp. 187–189.

[37] J. Metsa, M. Katara, and T. Mikkonen, "Comparing aspects with conventional techniques for increasing testability," in *IEEE International Conference on Software Testing, Verification, and Validation*, New York, 2008, pp. 387–395.

[38] A. Sioud, "Gestion de cycle de vie des objets par aspects pour C++," Master's thesis, UQaC, 2006.

[39] M. Wehrmeister, "An aspect-oriented model-driven engineering approach for distributed embedded real-time systems," Master's thesis, Federal University of Rio Grande do Sul, Brazil, 2009.

[40] J. Pesonen, M. Katara, and T. Mikkonen, "Production-testing of embedded systems with aspects," *Lecture Notes in Computer Science*, vol. 3875, pp. 90–102, 2006.

[41] A. A. A. Ghani and R. M. Parizi, "Aspect-oriented program testing: An annotated bibliography," *Journal of Software*, vol. 8, pp. 1281–1300, 2013.

[42] D. Sokenou and S. Herrmann, "Aspects for testing aspects?" in *1st Workshop on Testing Aspect-Oriented Programs at AOSD*, Chicago, USA, March 14-18 2005, pp. 1–6.

[43] Giladgar, "File:whiteboxtesting1.png," accessed July 1, 2017, https://commons.wikimedia.org/wiki/File:WhiteBoxTesting1.png CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0), via Wikimedia Commons.

[44] R. M. Jr., K. S. Trivedi, and P. R. M. Maciel, "Using accelerated life tests to estimate time to software aging failure," in *2010 IEEE 21st International Symposium on Software Reliability Engineering*, November 2010, pp. 211–219.

[45] M. Jain and D. Gopalani, "Memory leakage testing using aspects," in *2015 International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, 2015, pp. 436–440.

[46] S. D. Stoller, "Testing concurrent java programs using randomized scheduling," *Electronic Notes in Theoretical Computer Science*, vol. 70, pp. 142–157, 2002.

[47] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *IEEE transactions on software engineering*, vol. 27, 2001, pp. 99–123.

[48] M. Jain and D. Gopalani, "Use of aspects for testing software applications," in *Proc. Int. Conf. Advance Computing Conference (IACC)*, vol. 1, Bangalore, India, 2015, pp. 282–285.

[49] T. Aslam, "Aspectmatlab: An aspect-oriented scientific programming language," Master's thesis, School of Computer Science, McGill University, Montreal, 2010.

[50] C. Sharma and D. Karambir, "Optimization of basis path testing using genetic tabu search algorithm," *International Journal for Innovative Research in Science & Technology*, vol. 1, no. 11, pp. 489–492, 2015.

[51] D. H. Lee, S. Y. Kim, D. S. Choi, and H. G. Oh, "File fuzzing system using field information and fault-injection rule," *Journal of Security Engineering Research*, vol. 5, no. 6, pp. 497–508, 2008.

[52] S. B. Rajakumari and K. Umadevi, "A review on software fault injection methods and tools," *International Journal of Innovative Research in Computer and Communication Engineering*, vol. 03, pp. 1582–1587, 04 2015.

[53] N. Belblidia, M. Debbabi, A. Hanna, and Z. Yang, "AOP extension for security testing of programs," in *Canadian Conference on Electrical and Computer Engineering CCECE '06*, 2006, pp. 647–650.

[54] P. Baker, Z. R. Dai, J. Grabowski, O. Haugen, I. Schieferdecker, and C. Williams, *Model-Driven Testing: Using the UML Testing Profile*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

[55] C. Artho and A. Biere, "Advanced unit testing: How to scale up a unit test framework," in *Proceedings of the 2006 International Workshop on Automation of Software Test*, ser. AST'06, New York, USA, June 2006, pp. 92–98.

[56] N. Kuda, P. Gujjar Panduranga Rao, N. Kavita, and P. Chakka, "A study of the agile software development methods, applicability and implications in industry," *International Journal of Software Engineering and its Applications*, vol. 5, pp. 35–46, January 2011.

[57] H. Yu, D. Liu, G. Fan, and L. Chen, "A regression test technique for analyzing the functionalities of service composition," in *Software Engineering and Knowledge Engineering*, 2011, pp. 578–582.

[58] D. Gurpegui, *Net-C download*, accessed June 1, 2016], https://sourceforge.net/projects/netc/.

[59] Appwork GmbH, *jDownloader download*, accessed June 1, 2016, https://sourceforge.net/projects/jdownloader/.

[60] Deepak PK, *JScreenRecorder download*, accessed June 1, 2016, https://sourceforge.net/projects/jscreenrecorder/?source=directory.

[61] D. Gilbert, *JFreeChart download*, accessed June 1, 2016, https://sourceforge.net/projects/jfreechart/.

[62] ——, *JFreeChart Homepage*, accessed July 1, 2017, http://www.jfree.org/index.html.

[63] M. Mernik, *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*. IGI Global, 2012.

[64] P. Hudak, *Handbook of Programming Languages*. MacMillan, Indianapolis, 1998, vol. 3.

[65] M. Fowler, *Domain Specific Languages*, 1st ed. Addison-Wesley Professional, 2010.

[66] A. Fletcher, "File:learning curve diagram - steep and shallow, same functionality.jpg," accessed July 1, 2017, https://en.wikipedia.org/wiki/File:Learning_Curve_Diagram_--_Steep_and_Shallow,_Same_Functionality.jpg.

[67] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *Future of Software Engineering, 2007. FOSE '07*, May 2007, pp. 85–103.

[68] A. Ahonen, "Unit and integration testing of Java: JVM behavior-driven development testing frameworks vs. JUnit," Master's thesis, Aalto University, May 2017.

[69] S. Desai and A. Srivastava, *Software Testing: A Practical Approach*, 2nd ed. PHI Learning, January 2016.

[70] A. Jain, S. Sharma, S. Sharma, and D. Juneja, "Boundary value analysis for non-numerical variables: Strings," *Oriental Journal of Computer Science & Technology*, vol. 3, no. 2, pp. 323–330, 2010.

[71] A. Hussain, A. Razak, and E. Mkpojiogu, "The perceived usability of automated testing tools for mobile applications," *Journal of Engineering Science and Technology*, vol. 12, pp. 89–97, April 2017.

[72] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, "Understanding the test automation culture of app developers," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, April 2015, pp. 1–10.

[73] F. V. C. Ficarra, "Advances in new technologies, interactive interfaces and communicability," in *Proceedings of the First International Conference on Advances in New Technologies, Interactive Interfaces, and Communicability*, ser. ADNTIIC'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 1–7.

[74] S. L. Tsang, S. Clarke, and E. Baniassad, "An evaluation of aspect-oriented programming for Java-based real-time systems development," in *Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2004. Proceedings.*, May 2004, pp. 291–300.

[75] S. Gruner and J. van Zyl, "Software testing in small IT companies: A (not only) South African problem," *South African Computer Journal*, vol. 47, pp. 7–32, 2011.

[76] M. A. Fecko and C. M. Lott, "Lessons learned from automating tests for an operations support system," *Softw. Pract. Exper.*, vol. 32, no. 15, pp. 1485–1506, December 2002.

[77] D. Hoffman, "Cost benefits analysis of test automation," in *STARWEST 1999 - Software Testing Conference*, 1999, pp. 1–14.

[78] 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*, ser. ECOOP '01. London, UK, UK: Springer-Verlag, 2001, pp. 327–353.

[79] V. Dwarampudi, S. S. Dhillon, J. Shah, N. J. Sebastian, and N. Kanigicharla, "Comparative study of the pros and cons of programming languages Java, Scala, C++, Haskell, VB.NET, AspectJ, Perl, Ruby, PHP and Scheme - a team 11 COMP6411-S10 term report," *CoRR*, vol. abs/1008.3431, August 2010.

[80] Z. A. Barmi and A. H. Ebrahimi, "Automated testing of non-functional requirements based on behavioural scripts," Master's thesis, University of Gothenburg, Department of Computer Science and Engineering, December 2011.

[81] M. Fähndrich, M. Carbin, and J. R. Larus, "Reflective program generation with patterns," in *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, ser. GPCE '06. New York, NY, USA: ACM, 2006, pp. 275–284.

[82] S. Tyagi and P. Tarau, "A most specific method finding algorithm for reflection based dynamic Prolog-to-Java interfaces," in *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages*, ser. PADL'01, London, UK, 2001, pp. 322–336.

[83] Z. Shams and S. H. Edwards, "Reflection support: Java reflection made easy," *The Open Software Engineering Journal*, vol. 7, no. 1, pp. 38–52, 2013.

[84] T. Gendler, *Using AspectJ for Accessing Private Members without Reflection*, accessed June 1, 2016, http://blogs.vmware.com/vfabric/2012/04/using-aspectj-for-accessing-private-members-without-reflection.html.

[85] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely

invariants," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 35–45, December 2007.

[86] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, May 2002, pp. 291–301.

[87] M. Lippert and C. V. Lopes, "A study on exception detection and handling using aspect-oriented programming," in *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, 2000, pp. 418–427.

[88] J. Sedlacek and T. Hurka, *VisualVM: Home*, accessed July 1, 2017, https://visualvm.github.io/.

[89] Oracle, *HPROF: A Heap/CPU Profiling Tool*, accessed July 1, 2017, https://docs.oracle.com/javase/8/docs/technotes/samples/hprof.html.

[90] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Evaluating the accuracy of Java profilers," *SIGPLAN Not.*, vol. 45, no. 6, pp. 187–197, Jun. 2010.

[91] Apache Software Foundation, *Apache JMeter*, accessed July 1, 2017, http://jmeter.apache.org/.

[92] Micro Focus, *LoadRunner*, accessed July 1, 2017, https://software.microfocus.com/ko-kr/products/loadrunner-load-testing/pricing.

[93] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, "The Java language specification Java SE 7 edition," Addison-Wesley, 1997, oracle and/or its affiliates.

[94] B. Křena, Z. Letko, Y. Nir-Buchbinder, R. Tzoref-Brill, S. Ur, and T. Vojnar, "Runtime verification," S. Bensalem and D. A. Peled, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, ch. A Concurrency Testing Tool and Its Plug-Ins for Dynamic Analysis and Runtime Healing, pp. 101–114.

[95] S. Ritchie, *Pro .NET Best Practices*. Berkely, CA, USA: Apress, 2011.

[96] A. Deursen, L. M. Moonen, A. Bergh, and G. Kok, "Refactoring test code," Amsterdam, The Netherlands, Tech. Rep., 2001, http://www.ncstrl.org:8900/ncstrl/servlet/search?formname=detail&id=oai%3Ancstrlh%3Aercim_cwi%3Aercim.cwi%2F%2FSEN-R0119.

[97] L. Williams, G. Kudrjavets, and N. Nagappan, "On the effectiveness of unit test automation at Microsoft," in *2009 20th International Symposium on Software Reliability Engineering*, November 2009, pp. 81–89.

[98] K. Bareja and A. Singhal, "A review of estimation techniques to reduce testing efforts in software development," in *2015 Fifth International Conference on Advanced Computing Communication Technologies*, February 2015, pp. 541–546.

[99] P. G. Frankl and S. N. Weiss, "An experimental comparison of the effectiveness of branch testing and data flow testing," *IEEE Transactions on Software Engineering*, vol. 19, no. 8, pp. 774–787, August 1993.

[100] M. J. Rutherford, A. Carzaniga, and A. L. Wolf, "Evaluating test suites and adequacy criteria using simulation-based models of distributed systems," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 452–470, July 2008.

[101] T. W. Williams, M. R. Mercer, J. P. Mucha, and R. Kapur, "Code coverage, what does it mean in terms of quality?" in *Annual Reliability and Maintainability Symposium. 2001 Proceedings. International Symposium on Product Quality and Integrity (Cat. No.01CH37179)*, 2001, pp. 420–424.

[102] G. J. Myers and C. Sandler, *The Art of Software Testing.* John Wiley & Sons, 2004.

[103] K. Meffert, N. Rotstan, C. Knowles, and U. B. Sangiorgi, *JGAP: Java Genetic Algorithms Package download*, accessed March 1, 2017, http://jgap.sourceforge.net/.

[104] D. R. Kuhn and V. Okun, "Pseudo exhaustive testing for software," in *30th Annual IEEE Software Engineering Workshop*, 2006, pp. 153–158.

[105] M. Fewster and D. Graham, *Software Test Automation.* Addison-Wesley, 1999.

[106] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *Future of Software Engineering, 2007. FOSE '07*, May 2007, pp. 85–103.

[107] M. Jain and D. Gopalani, "Aspect oriented programming and types of software testing," in *2016 Second International Conference on Computational Intelligence Communication Technology (CICT)*, February 2016, pp. 64–69.

[108] K. Chen and C. Chien, "Extending the field access pointcuts of AspectJ to arrays," *Journal of Software Engineering Studies*, vol. 2, pp. 93–102, January 2007.

[109] B. Harbulot and J. R. Gurd, "A join point for loops in AspectJ," in *Proceedings of the 5th International Conference on Aspect-oriented Software Development*, ser. AOSD '06. New York, NY, USA: ACM, 2006, pp. 63–74.

Appendix A

Source code for the *ChartPanel* class of *JFreeChart*

Listing A.1: JFreeChart: *paintComponent* method leaks memory

```
public void paintComponent(Graphics g) {
        super.paintComponent(g);
        if (this.chart == null) {
                return;
        }

        Graphics2D g2 = (Graphics2D) g.create();
        // first determine the size of the chart rendering area...
        Dimension size = getSize();
        Insets insets = getInsets();
        Rectangle2D available = new Rectangle2D.Double(insets.left,
            ↪ insets.top,
        size.getWidth() - insets.left - insets.right,
        size.getHeight() - insets.top - insets.bottom);

        boolean scale = false;
        double drawWidth = available.getWidth();
        double drawHeight = available.getHeight();
        this.scaleX = 1.0;
        this.scaleY = 1.0;
        if (drawWidth < this.minimumDrawWidth) {
                this.scaleX = drawWidth / this.minimumDrawWidth;
                drawWidth = this.minimumDrawWidth;
                scale = true;
        }
```

JFreeChart: *paintComponent* method leaks memory (contd.)

```
        else if (drawWidth > this.maximumDrawWidth) {
                this.scaleX = drawWidth / this.maximumDrawWidth;
                drawWidth = this.maximumDrawWidth;
                scale = true;
        }
        if (drawHeight < this.minimumDrawHeight) {
                this.scaleY = drawHeight / this.minimumDrawHeight;
                drawHeight = this.minimumDrawHeight;
                scale = true;
        }
        else if (drawHeight > this.maximumDrawHeight) {
                this.scaleY = drawHeight / this.maximumDrawHeight;
                drawHeight = this.maximumDrawHeight;
                scale = true;
        }
        Rectangle2D chartArea = new Rectangle2D.Double(0.0, 0.0,
            ↪ drawWidth, drawHeight);
        if (this.useBuffer) {
                if ((this.chartBuffer == null) || (this.chartBufferWidth !=
                    ↪ available.getWidth()) || (this.chartBufferHeight !=
                    ↪ available.getHeight())) {
                        this.chartBufferWidth = (int) available.getWidth();
                        this.chartBufferHeight = (int) available.getHeight();
                        GraphicsConfiguration gc =
                            ↪ g2.getDeviceConfiguration();
                        this.chartBuffer = gc.createCompatibleImage(this.
                            ↪ chartBufferWidth, this.chartBufferHeight,
                            ↪ Transparency.TRANSLUCENT);
                        this.refreshBuffer = true;
                }
                if (this.refreshBuffer) {
                        this.refreshBuffer = false; // clear the flag
                        Rectangle2D bufferArea = new Rectangle2D.Double(0,
                            ↪ 0, this.chartBufferWidth,
                            ↪ this.chartBufferHeight);
                        Graphics2D bufferG2 = (Graphics2D)
                        this.chartBuffer.getGraphics();
                        Composite savedComposite = bufferG2.getComposite();
                        bufferG2.setComposite(AlphaComposite.getInstance(
                            ↪ AlphaComposite.CLEAR, 0.0f));
                        Rectangle r = new Rectangle(0, 0,
                            ↪ this.chartBufferWidth, this.chartBufferHeight);
```

JFreeChart: *paintComponent* method leaks memory (contd.)

```
                        bufferG2. fill (r);
                        bufferG2.setComposite(savedComposite);
                        if (scale) {
                                AffineTransform saved =
                                    ↪ bufferG2.getTransform();
                                AffineTransform st =
                                    ↪ AffineTransform.getScaleInstance(
                                    ↪ this.scaleX, this.scaleY);
                                bufferG2.transform(st);
                                this.chart.draw(bufferG2, chartArea,
                                    ↪ this.anchor, this.info);
                                bufferG2.setTransform(saved);
                        }
                        else {
                                this.chart.draw(bufferG2, bufferArea,
                                    ↪ this.anchor, this.info);
                        }
                }
                g2.drawImage(this.chartBuffer, insets.left , insets.top, this);
        }
        else {
                AffineTransform saved = g2.getTransform();
                g2.translate(insets.left , insets.top);
                if (scale) {
                        AffineTransform st =
                            ↪ AffineTransform.getScaleInstance(this.scaleX,
                            ↪ this.scaleY);
                        g2.transform(st);
                }
                this.chart.draw(g2, chartArea, this.anchor, this.info);
                g2.setTransform(saved);
        }
        Iterator iterator = this.overlays.iterator();
        while (iterator.hasNext()) {
                Overlay overlay = (Overlay) iterator.next();
                overlay.paintOverlay(g2, this);
        }
        drawZoomRectangle(g2, !this.useBuffer);
        g2.dispose();
        this.anchor = null;
        this.verticalTraceLine = null;
        this.horizontalTraceLine = null;
}
```

Appendix B

Important tokens generated by the lexical analyser

Table B.1: Important tokens returned by lexer to the yacc parser

Token	**Relevance**
COMMBLOCK	A block of 4 continuous "/", i.e., "////" found
COLON	When a colon separating the *itemname* from *itemdescription* is found
TYPE	This token indicates the type of testing that has to be performed by the generated aspect
CLASSNAMETAG	When the *itemname classname* is found
CLASSNAME	For the class name provided by the tester to be used for conducting the test
NAME	When the tester has provided the name for the generated testing aspect through *itemname aspectname*
ASPECTNAME	For the name that has been provided for the generated testing aspect
FILEPATHTOK	When the *itemname filepath* is found
FILEPATHNAME	When the file name provided by tester for the purpose of conducting the test is found
METHODSIGNATURETAG	When the *itemname methodsignature* is found
ACCESSSPECIFIER	When one of the access specifiers *public, protected, private* are found

Table B.1 : Important tokens returned by lexer to the yacc parser, continued

Token	**Relevance**
RETURNTYPE	For the return type of the method given by the tester
METHODNAME	For the name of the method given by the tester
METHODARGTYPE	For the type of all the arguments of the method given by the tester
METHODARGNAME	For all the arguments' names of the method given by the tester
TESTARGTOKEN	When the *itemname argumentname* is found
TESTARGNAME	For every method argument to be used for conducting the test
VALUE	For values provided by the tester to be used for testing; for example, for different values of various method's arguments to be used for conducting the test
BBTSETUPTOKEN	When the *itemname setup* is found
EXPTOKEN	When the *itemname expected* is found
EXPVALUE	For expected values provided by the tester to be used for matching the test results
ARG	For the arguments whose values have to be kept fixed while testing
VAL	For the fixed value to be used for the specified argument
FUZZTYPETOK	When the *itemname fuzztype* is found
FUZZTYPE	For the type of fuzz testing: *replace, overwrite, insertafter, insertbefore*
FUZZLOCATIONTOK	When the *itemname fuzzlocation* which is used to indicate the location to be used for fuzzing is found
FUZZLOCATIONVALUE	When the location to be used for fuzzing is found
FUZZVALUETOK	When the *itemname fuzzvalue* is found
FUZZVALUES	For every value provided by the tester to be used for fuzzing

Table B.1 : Important tokens returned by lexer to the yacc parser, continued

Token	**Relevance**
THREADNAME	For the name of the thread that has to be tested for concurrent errors
INSERTNOISETOK	When the *itemname insertnoise* is found
INSNOISEVALUE	For before/after condition given by the tester for noise injection in concurrency testing
PROBTOK	When the *itemname probabilitypercentage* is found
PROBVALUE	For the value of probability (in %age) given by the tester in concurrency testing
SLEEPTOK	When the *itemname sleep* is found
SLEEPVALUE	For the value of sleep (random or fixed integer) provided by the tester
SHAREDVARNAME	For the name of the shared variable to be used for concurrency testing
INCLASSNAME	For the inner class name used in memory leakage testing
OUTCLASSNAME	For the outer class name used in memory leakage testing
LTSETUPTOKEN	When the *itemname loadtestinginitialsetup* is found
CLASSLOADNAME	For the name of the class provided by the tester whose dummy objects are to be created for load testing
NOOFOBJCOUNT	For the number of dummy objects that are to be created for load testing
PARAMETERNAME	For every form parameter that has to be tested in servlet testing

Appendix C

TAGL Grammar

Listing C.1: TAGL Grammar

```
S       :       MLTANDNPETDENOTATION
                |
                MLTCASEIIDENOTATION
                |
                BBTDENOTATION
                |
                LTDENOTATION
                |
                CTDENOTATION
                |
                FTDENOTATION
                |
                STDENOTATION
        ;

MLTANDNPETDENOTATION : ASPECTTYPESTMT ASPECTNAMESTMT CLASSNAMESTMT
                       ;

MLTCASEIIDENOTATION     :       ASPECTTYPESTMT ASPECTNAMESTMT
                                INNERCLASSSTMT METHODSIGSTMT
                                OUTERCLASSSTMT
                        ;

BBTDENOTATION    :      ASPECTTYPESTMT ASPECTNAMESTMT CLASSNAMESTMT
                        METHODSIGSTMT TESTARGNAMESTMT VALUESSTMT
                 |
                        ASPECTTYPESTMT ASPECTNAMESTMT CLASSNAMESTMT
                        METHODSIGSTMT TESTARGNAMESTMT VALUESSTMT
                        EXPVALUESSTMT
                 |
                        ASPECTTYPESTMT ASPECTNAMESTMT CLASSNAMESTMT
                        METHODSIGSTMT BBTSETUPSTMT TESTARGNAMESTMT
                        VALUESSTMT EXPVALUESSTMT
                 |
```

```
                    ASPECTTYPESTMT ASPECTNAMESTMT CLASSNAMESTMT
                    METHODSIGSTMT TESTARGNAMESTMT NOMINALSTMT
                    VALUESSTMT EXPVALUESSTMT
                ;

LTDENOTATION    :   ASPECTTYPESTMT ASPECTNAMESTMT CLASSNAMESTMT
                    METHODSIGSTMT CLASSLOADSTMT NOOFOBJSTMT
                |
                    ASPECTTYPESTMT ASPECTNAMESTMT CLASSNAMESTMT
                    METHODSIGSTMT CLASSLOADSTMT NOOFOBJSTMT
                    LTSETUPSTMT
                ;

CTDENOTATION    :   ASPECTTYPESTMT ASPECTNAMESTMT THREADNAMESTMT
                    SHAREDVARINSNOISEPROBSTMTS SLEEPSTMT
                ;

FTDENOTATION    :   ASPECTTYPESTMT ASPECTNAMESTMT CLASSNAMESTMT
                    METHODSIGSTMT FILENAMESTMT FUZZTYPEANDLOCSTMTS
                    FUZZVALUESTMT
                ;

STDENOTATION    :   ASPECTTYPESTMT ASPECTNAMESTMT
                    PARAMETERNAMESSTMT VALUESSTMT
                ;

SHAREDVARINSNOISEPROBSTMTS  :   SHAREDVARSTMT INSNOISESTMT
                            |   INSNOISESTMT PROBSTMT
                            ;

ASPECTTYPESTMT  :   COMMBLOCK TYPE COLON ASPECTTYPE
                ;

ASPECTNAMESTMT  :   COMMBLOCK NAME COLON ASPECTNAME
                ;

CLASSNAMESTMT   :   COMMBLOCK CLASSNAMETAG COLON CLASSNAME
                ;

INNERCLASSSTMT  :   COMMBLOCK INCLASSNAMETAG COLON INCLASSNAME
                ;

OUTERCLASSSTMT  :   COMMBLOCK OUTCLASSNAMETAG COLON OUTCLASSNAME
                ;

METHODSIGSTMT   :   COMMBLOCK METHODSIGNATURETAG COLON
                    METHODSIGNATURE
                ;

METHODSIGNATURE : ACCESSSPECIFIER RETURNTYPE METHODNAME METHODARGLISTS
                ;

METHODARGLISTS  :
                |   METHODARGLISTS METHODARGLIST
                ;
```

```
METHODARGLIST   :   METHODARGTYPE METHODARGNAME
                ;

BBTSETUPSTMT    :   COMMBLOCK BBTSETUPTOKEN COLON SETUPMETHODS
                ;

SETUPMETHODS    :   METHODSIGNATURE
                |   SETUPMETHODS METHODSIGNATURE
                ;

TESTARGNAMESTMT :   COMMBLOCK TESTARGTOKEN COLON ARGLIST
                |   COMMBLOCK TESTARGTOKEN COLON FIXVALUEARGLIST
                    ARGLIST
                |   COMMBLOCK TESTARGTOKEN COLON ARGLIST
                    FIXVALUEARGLIST
                ;

FIXVALUEARGLIST :   FIXVALUEARGTEXT
                |   FIXVALUEARGLIST FIXVALUEARGTEXT
                ;

FIXVALUEARGTEXT :   ARG '=' VAL
                ;

NOMINALSTMT     :   COMMBLOCK NOMINALTOKEN COLON NOMINALARGLIST
                ;

NOMINALARGLIST  :   NOMINALVALUEARGTEXT
                |   NOMINALARGLIST NOMINALVALUEARGTEXT
                ;

NOMINALVALUEARGTEXT :       ARG '=' VAL
                    ;

ARGLIST         :   TESTARGNAME
                |   ARGLIST TESTARGNAME
                ;

CLASSLOADSTMT   :   COMMBLOCK CLASSLOADTOK COLON CLASSLOADNAME
                ;

NOOFOBJSTMT     :   COMMBLOCK NOOFOBJTOK COLON NOOFOBJCOUNT
                ;

LTSETUPSTMT     :   COMMBLOCK LTSETUPTOKEN COLON JAVASTMTS
                ;

THREADNAMESTMT  :   COMMBLOCK THREADNAMETOK COLON THREADNAME
                ;

SHAREDVARSTMT   :   COMMBLOCK SHAREDVARTOK COLON SHAREDVARNAME
                ;

INSNOISESTMT    :   COMMBLOCK INSERTNOISETOK COLON INSNOISEVALUE
                ;
```

```
PROBSTMT    :       COMMBLOCK PROBTOK COLON PROBVALUE
            ;

SLEEPSTMT   :       COMMBLOCK SLEEPTOK COLON SLEEPVALUE
            ;

PARAMETERNAMESSTMT  :       COMMBLOCK PARAMETERSNAMETOKEN COLON
                            PARAMETERLIST
                    ;

PARAMETERLIST       :
                    |       PARAMETERLIST PARAMETERNAME
                    ;

VALUESSTMT  :       COMMBLOCK VALUESTOKEN COLON VALUELIST
            ;

VALUELIST   :
            |       VALUELIST VALUE
            ;

EXPVALUESSTMT       :       COMMBLOCK EXPTOKEN COLON EXPVALUELIST
                    ;

EXPVALUELIST        :
                    |       EXPVALUELIST EXPVALUE
                    ;

FILENAMESTMT        :       COMMBLOCK FILEPATHTOK COLON FILEPATHNAME
                    ;

FUZZTYPEANDLOCSTMTS :       FUZZLOCATIONSTMT
                    |       FUZZTYPESTMT FUZZLOCATIONSTMT
                    ;

FUZZTYPESTMT        :       COMMBLOCK FUZZTYPETOK COLON FUZZTYPE
                    ;

FUZZLOCATIONSTMT :      COMMBLOCK FUZZLOCATIONTOK COLON
                        FUZZLOCATIONVALUES
                 ;

FUZZLOCATIONVALUES  :
                    |       FUZZLOCATIONVALUES FUZZLOCATIONVALUE
                    ;

FUZZVALUESTMT       :       COMMBLOCK FUZZVALUETOK COLON FUZZVALUES
                    ;
```

Brief bio-data

I, Manish Jain, born on 25th-July-1981 received the B.E. degree in Computer Science and Engineering from Government Engineering College, Bikaner with honours. I completed my M.Tech. from Malaviya National Institute of Technology from the Computer Science and Engineering Department in the year 2012 with 8 CGPA.

After doing a brief service in the IT industry, I joined Baldev Ram Mirdha Institute of Technology in the year 2005 as an Assistant Professor and am continuing with the same organization till date. Working closely with the management at my current organization, I have taken up all sorts of challenges and responsibilities including teaching and administrative tasks of varied nature.

I am proficient at grasping new technical concepts quickly and utilise the same in a productive manner. I have got a good knowledge of programming languages like .Net, C, C++, Java, AspectJ, Eiffel, Lex and Yacc, Latex etc. I have developed software projects like ERP and Time Management System which are running live at my current organization. I am skilled to accomplish projects with minimum resources and meeting stringent deadlines with incredible standards.

This thesis has been written in the fulfilment of the requirements for the degree of Doctor of Philosophy from the department of Computer Science and Engineering, MNIT-Jaipur (2013-2018). The list of publications as a part of this research work are enlisted here under:

- M. Jain and D. Gopalani, "Use of aspects for testing software applications," IEEE International Advance Computing Conference (IACC), Bangalore, India, 2015, pp. 282-285. doi: 10.1109/IADCC.2015.7154714

- M. Jain and D. Gopalani, "Memory leakage testing using aspects," International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT), Davangere, India, 2015, pp. 436-440. doi: 10.1109/ICATCCT.2015.7456923

- M. Jain and D. Gopalani, "Aspect Oriented Programming and Types of Software Testing," Second International Conference on Computational Intelligence and Communication Technology (CICT), Ghaziabad, India, 2016, pp. 64-69. doi: 10.1109/CICT.2016.22

- M. Jain and D. Gopalani, "Testing Application Security with Aspects," International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT), Chennai, India, 2016, pp. 3161-3165. doi: 10.1109/ICEEOT.2016.7755285

- Accepted: M. Jain and D. Gopalani, "Domain Specific Language for Automatically Generating Testing Aspects," IEEE International Conference on Emerging Trends in Computing and Communication Technologies (ICETCCT), Dehradun, India, 2017.

- M. Jain and D. Gopalani, "Automated Java Testing: JUnit versus AspectJ," International Journal of Computer and Systems Engineering: International Science Index, Volume 11:11, 2017, pp. 1153-1158. dai:10.1999/1307-6892/100082245