World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:11, No:11, 2017

Automated Java Testing: JUnit versus AspectJ

Manish Jain, Dinesh Gopalani

***Abstract*—Growing dependency of mankind on software technology increases the need for thorough testing of the software applications and automated testing techniques that support testing activities. We have outlined our testing strategy for performing various types of automated testing of Java applications using AspectJ which has become the de-facto standard for Aspect Oriented Programming (AOP). Likewise JUnit, a unit testing framework is the most popular Java testing tool. In this paper, we have evaluated our proposed AOP approach for automated testing and JUnit on various parameters. First we have provided the similarity between the two approaches and then we have done a detailed comparison of the two testing techniques on factors like lines of testing code, learning curve, testing of private members etc. We established that our AOP testing approach using AspectJ has got several advantages and is thus particularly more effective than JUnit.

***Keywords*—Aspect oriented programming, AspectJ, Aspects, JUnit, software testing.

I. Introduction

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.

In this direction in order to facilitate the process of software testing, various automated testing tools have been developed by the researchers and developers. For example, for the testing of Java applications, we have number of testing tools available like JUnit, TestNG, Mockito, Selenium, Arquillian, JMeter etc. However, the most popularly recognised automated testing tool for testing Java applications is JUnit [1], [2]. Moreover in our previous papers [3]-[6], we have established the use of aspects in AspectJ, which has become the de-facto standard for Aspect Oriented Programming (AOP) [7], for the purpose of carrying out different types of testing of Java applications. We have also used aspects in AspectJ to test well known open source Java applications like Netc, JFreeChart, JDownloader, JGAP etc. and detected remarkable bugs into them.

AOP is a new methodology that provides a mechanism for separation of crosscutting concerns from the core concern. A concern is actually a functionality necessary in a software system. Any software system is thus a realisation of one or more concerns. For e.g. for a 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 which are 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. In AOP, a new unit of modularisation - an ***aspect*** - is introduced within which we implement the crosscutting concerns instead of fusing them into the core modules. In our proposed methodology, we used these aspects for the purpose of test automation.

Our AOP based methodology can be used to automate the generation of test cases, write the test script, execute the test cases and further compare the results with the expected results and prepare a test report. Using aspect oriented languages, the testing code can be written in the form of *before*, *after* or *around* advices within the aspects. The aspect weaver weaves the testing code with the source code under test as shown in Fig. 1. 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 test report giving details about the failed and successful tests is prepared.

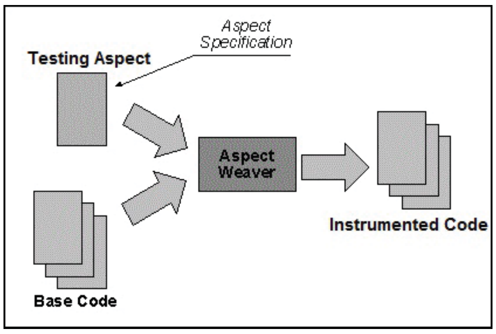

Fig. 1 Role of Aspect Weaver

On the other hand, JUnit which is the most popular unit testing framework for Java applications has played a very important role in the test-driven development. JUnit belongs to xUnit family of unit-testing frameworks [8] which are used for developing and executing unit test cases, and for regression testing. JUnit has been designed for the purpose of writing and running tests in Java and to ensure that the code is validated

Manish Jain is with the Department of Computer Science, Malaviya National Institute of Technology, Jaipur, India (e-mail: halomanish@gmail.com).
Dinesh Gopalani is with the Department of Computer Science, Malaviya National Institute of Technology, Jaipur, India (e-mail: dgopalani.cse@mnit.ac.in).

and functions as per the requirement specification. For testing a piece of code using JUnit, a class which extends the *TestCase* class is created and then the various test methods are added to it. JUnit is annotation based. Annotations are actually syntactic metadata that are added to the testing code for achieving better readability and structure.

In this paper, we shall discuss the resemblances between AspectJ and JUnit which make AspectJ a compelling tool for performing testing of Java applications. Further, we shall compare the two approaches and establish the benefits of using our AOP based approach. We have used AspectJ version 1.8.10 and JUnit 4 in our work.

The paper is organised as follows: in Section II, we provide a list of the related work done in this field. In Section III, we have discussed the resemblances between AspectJ and JUnit frameworks of testing. Section IV provides a comparative analysis of the two approaches on various parameters. At the end, Section V is used to specify the conclusion and future work. References are enumerated at last.

II. Related Work

Li and Xie [9] in their paper have claimed that aspects make good stubs and drivers. We evaluated this assertion and used this property of aspects for performing integration testing. JUnit though most popular Java testing tool does not have necessary features for performing integration testing of Java applications. Duclos et al. [10] used AspectC++ for carrying out certain basic testings of C++ programs. Sioud [11] implemented the missing garbage collection in C++ using AspectC++. Java has its own garbage collection mechanism in place but still there are possibilities of memory leakages like buffer overflows and null pointer exceptions which we could test using AspectJ. Sokenou and Herrmann [12] used AOP to test programs written in AOP languages. They stated that aspects seem worthy for testing the aspect-oriented systems. Copty et. al [13] have used AspectJ to implement the certain functionalities of the concurrency testing tool ConTest. We enhanced their idea by using aspects to discover bugs like race conditions or deadlocks which generally occur in concurrency based programs. Pesonen et al. [14] applied aspect orientation to the production testing framework for Symbian OS which contained embedded programs only. We experimented with aspects to use these for testing non-embedded Java programs. Moreover, in this paper we have established various benefits of using AspectJ over JUnit for testing Java applications.

III. Resemblance of AspectJ with JUnit

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 is quite a class-like concept [15].

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 a test case. In AspectJ, the same functionality is achieved by the *around* advice in which the method to be tested can be instrumented & tested with desired input values.

```
@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);
}
```

Fig. 2 Testing a method in Student Class using JUnit

The annotations *@Before* and *@BeforeClass* in JUnit indicate the methods which setup the necessary pre-conditions 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 shown in Figs. 2 and 3. 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 before the test class. To achieve the same functionality, we use the *within* pointcut with the testing aspect name along with a before advice that shall capture all the joinpoints within the scope of the testing aspect and execute the code written inside advice before these.

Likewise, JUnit annotations *@After* and *@AfterClass* are used to indicate the methods which gets executed after execution of the tests methods and perform certain cleanup tasks like delete temporary variables, reset variable, disconnect from database etc. These annotations can be directly mapped onto the *after* advices available in AspectJ along with appropriate pointcuts, on the same lines as discussed for *@Before* and *@BeforeClass*.

The annotations *@Before, @BeforeClass, @After, @AfterClass, @Test* are the most important annotations in JUnit which form the basis of writing testing code in JUnit. Likewise, the *before, after* and *around* advices in AspectJ are the most important action and decision part that form the dynamic crosscutting rules [16].

The functionality of *@Ignore* annotation in JUnit can be implemented in AspectJ by adding a simple "&& if(false)" to the pointcut so that the corresponding advice shall not be executed [16] as shown hereunder:

pointcut selectedJoinpoints() : within(package.*) && if(false);

Thus, as discussed above, all the annotations in JUnit can

```
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!=6.0)
            System.out.println("Error");
      return proceed(st); //do original processing
}
```

Fig. 3 Testing a method in Student Class using AspectJ

be equated with one of the available constructs in AspectJ. In view of the same, all the functionality and the types of testing of Java applications that can be carried out using JUnit are equally possible with AspectJ. Moreover, both AspectJ and JUnit can be easily integrated with Eclipse, the development environment for Java. AspectJ Development Tools for Eclipse (AJDT) provides the requires tooling support to develop and run AspectJ applications on Eclipse [17]. Similarly, JUnit plug-in is available for Eclipse which comes built in with most of the latest versions of Eclipse [18].

IV. Comparison of AspectJ and JUnit Frameworks of Automated Testing

As we have explained in our previous papers [3]-[6], AspectJ has all necessary features and constructs that make it suitable to be used as a testing framework for testing Java applications. In this section, we shall outline a thorough comparison of our proposed testing approach with JUnit, which is the most popular testing tool for Java applications. In order to compare the two approaches, we shall mainly focus on parameters like lines of testing code, possibility of carrying out various types of testing, testing of fields with private access, learning curve and others.

A. Lines of Testing Code

During our research, we observed that using aspects in AspectJ 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, 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 whereas the same required 34 lines of code when written using JUnit. A comparison of number of lines of testing code for testing of three different methods is shown in Fig. 4. It is evident from Fig. 4 that use of AspectJ reduces the number of lines of testing code and thus save the tester's valuable time which can further accelerate the bug discovery.

We would also like to state here that 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

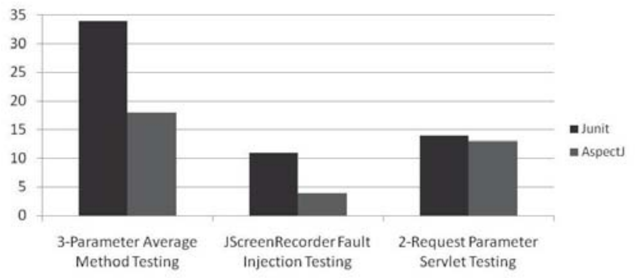

Fig. 4 Number of lines of testing code is reduced using AspectJ

any method regardless of 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. Similarly, 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 which increases the number of lines of code.

B. Testing Private Members

JUnit does not provide upfront mechanism for testing the private methods. On the other hand, when a private method contains an algorithm which requires more unit testing than it is possible through the public interfaces, then it becomes practically important to test the private method as well. 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 by the test class. At times 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. In such cases, it becomes necessary to directly test the operations of the private method.

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, the test code becomes verbose when the reflection API is deployed. Using Reflection produces test code

```
String username[]={"tom","john","peter"};
String password[]={"12345","abcd","abc"};
pointcut stub(): call(* login.databaseModule(..));
Object around(): stub()
{
      Object args[]=thisJoinPoint.getArgs();
      String s=(String)args[0];
      for(i=0;i < username.length;i++)
      {
            if(s.equals(username[i]))
            {
                  return (String)password[i];
            }
      }
      return null;
}
```

Fig. 5 Stub created using AspectJ

which is harder to understand and maintain. 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 [19]. In fact, using reflection mechanism is not recommended by most of the software developers [20]. However using AspectJ, the private methods can be easily accessed in the aspect for testing by adding *privileged* keyword to the aspect. Code inside privileged aspects has access to all members of the captured object, even the private ones.

In a nutshell, by definition unit test is intended to test every unit of code which should be irrespective of its scope and since AspectJ has provisions for accessing the public as well as private components of the class within the testing aspects, therefore it is a better choice for performing unit tests.

C. Performing Integration Testing

JUnit is suitable for conducting unit tests only but using AspectJ we can carry out other types of testing as well. Using aspects in AspectJ, we can create a stub or driver in lieu of an application module which is either not fully developed yet or needs extensive resources for execution. Such a stub or driver is useful for performing Integration Testing. AspectJ provides us with around advice which can be used to completely bypass the execution of the captured joinpoint and thus around advice can be utilised to write the functionality of a missing module to be integrated or a light-weighted alternative of a module. However, JUnit has limited support for Integration Testing [21].

In order to understand how a stub can be created using aspects in AspectJ, let us take example of a *login module*. Suppose the login module depends on a backend database module which checks for the user id value passed to it by the login module in its database and returns the password on a match (null otherwise). Now suppose the login module is ready and we want to test it, but the database module and the associated database is not ready. In such a case, a stub can be written in the form of aspect which shall mock the database module and can be used to return suitable value to the login module. The code snippet in Fig. 5 throws some light on our idea.

D. Performing 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. We used pointcuts in AspectJ 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 can be tested both at compile time as well as run time. JUnit has got no provision for testing the invariant conditions.

E. Performing Servlet Testing

JUnit alone does not suffice to test a Java servlet application. During unit 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. On the other hand, when we tested Java servlets using our AspectJ approach, we simply used *javax.servlet* package which is a part of the Java Enterprise Edition. Thus, servlet testing using AspectJ is straightforward and does not involve the use of any external API.

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. The servlet testing aspect shown in Fig. 6 implements the Filter interface and then creates an object of the *RequestWrapper* class within the *doFilter()* method.

F. Learning Curve

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 & understand whereas the corresponding aspect code is quite uncomplicated. The AspectJ testing code shown in Fig. 7 that tests the *getAverage* method of the *Student* class substantiates our point.

V. Conclusion and Future Work

In our earlier research work our intent was to find out whether AspectJ is suitable for testing Java applications and further to determine which all type of software testing is possible using aspects in AspectJ. We carried out various types of testing of selected widely used Java software from the open source community like jGnash, NetC, JFreeChart, JDownloader, JGAP etc. using aspects.

```
public aspect AspectA implements Filter
{
      public void doFilter(ServletRequest request,ServletResponse response,FilterChain chain)
      throws IOException,ServletException
      {
            //RequestWrapper 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
      }
}
```

Fig. 6 Servlet testing aspect using AspectJ

```
public aspect multipleInputValues
{
      before() : execution(public static void main(String[]))
      {
            int[] mark1 = {10,20,30,40,50,60,71,80,90,91};
            int[] mark2 = {10,20,30,40,50,60,71,80,90,91};
            int[] mark3 = {10,20,30,40,50,60,71,80,90,91};

            int i=0;
            for (i=0;i<10;i++)
            {
                  Student s = new Student();
                  s.setMarks1(mark1[i]);
                  s.setMarks2(mark2[i]);
                  s.setMarks3(mark3[i]);

                  double result = s.getAverage();
                  validateResult(i,result);
            }

            System.exit(0);
      }
}
```

Fig. 7 Testing with multiple input values

In this paper, we have established the benefits of using AspectJ for software testing over the conventional techniques. The number of test cases for testing bigger projects are too high [22] and practically it is quite time consuming to test the software with all the test cases using the conventional testing techniques. Although, with AspectJ we can capture multiple execution points in the code using wild cards in pointcuts and therefore test case execution consumes lesser time. Moreover, it had been yet another challenge in software testing using conventional techniques to select the code to be included in a test adequacy criterion. Aspects in AspectJ extend our ability to select the code in accordance with the intent of the tester. As far as the available automated testing tools are concerned, to use these tools, testers need skills like knowledge of test tools, general software, domain and system knowledge etc [23]. Aspects, on the other hand, are easier to be adopted into existing development projects.

In essence, in our paper we have established the use of aspects in AspectJ to perform various kind of software testing and listed their benefits as well.

There are several lines of experimentation which arise from our research work which can be carried out in future. Our AspectJ approach can be extended to cover other testing types like concurrency testing, regression testing, loop testing etc. Moreover, since AspectJ is a new programming paradigm and not all developers or testers are familiar with this technology, a Domain Specific Language (DSL) can be created whose syntax is natural language-like and the statements written thereof are automatically converted to testing aspects using the DSL parser so that even the testers who do not have the knowledge of AspectJ can still avail the benefits of testing using AspectJ. The preliminary results obtained in this direction are encouraging.

REFERENCES

[1] 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, 04 2017.

[2] 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.

[3] M. Jain and D. Gopalani, "Use of aspects for testing software applications," in *2015 IEEE International Advance Computing Conference*, June 2015, pp. 282–285.

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

[5] M. Jain and D. Gopalani, "Aspect oriented programming and types of software testing," in *2016 Second International Conference on Computational Intelligence Communication Technology*, Feb 2016, pp. 64–69.

[6] M. Jain and D. Gopalani, "Testing application security with aspects," in *2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT)*, March 2016, pp. 3161–3165.

[7] F. V. C. Ficarra, C. de Castro Lozano, M. P. Jiménez, E. Nicol, A. Kratky, and M. Cipolla-Ficarra, *Advances in New Technologies, Interactive Interfaces, and Communicability*. Springer, 2011.

[8] A. Z. Javed, "Model-driven framework for context-dependent testing of components," Ph.D. dissertation, School of Information Technology and Electrical Engineering, The University of Queensland, August 2007.

[9] X. Li and X. Xie, "Research of software testing based on AOP," in *IEEE 3rd International Conference on Intelligent Information Technology Application*, vol. 1, 2009, pp. 187–189.

[10] 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.

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

[12] D. Sokenou and S. Herrmann, "Aspects for testing aspects?" in *1st Workshop on Testing Aspect-Oriented Programs*, 2005.

[13] S. Copty and S. Ur, "Multi-threaded testing with AOP is easy, and it finds bugs!" *Lecture Notes in Computer Science*, vol. 3648, pp. 740–749, 2005.

[14] 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.

[15] 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.

[16] R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming*. Greenwich, USA: Manning Publications Co., 2003.

[17] D. Gotseva and M. Pavlov, "Aspect-oriented programming with AspectJ," *International Journal of Computer Science Issues*, pp. 212–218, 2012.

[18] P. Bouillon, M. Burger, and A. Zeller, "Automated debugging in Eclipse: (at the touch of not even a button)," in *Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology eXchange*, ser. Eclipse '03. New York, USA: ACM, 2003, pp. 1–5.

[19] S. Tyagi and P. Tarau, "A most specific method finding algorithm for reflection based dynamic prolog-to-java interfaces," in *PADL*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 322–326.

[20] Z. Shams, "Reflection support: Java reflection made easy," *The Open Software Engineering Journal*, vol. 7, pp. 38–52, 01 2014.

[21] 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: ACM, 2006, pp. 92–98.

[22] D. R. Kuhn and V. Okun, "Pseudo exhaustive testing for software," in *30th Annual IEEE Software Engineering Workshop*, 2006, pp. 153–158.

[23] D. Rafi, K. Moses, K. Petersen, and M. Mantyla, "Testing non-functional requirements with aspects," in *IEEE 7th International Workshop on Automation of Software Test AST*, 2012, pp. 36–42.