Towards a Practical Approach to Testing Pointcut Descriptors With JQuati

Emmanuel de Castro Santana*, Sandro Hiroki Tanaka*, Eduardo Martins Guerra*, Clovis Torres Fernandes†, Fábio Fagundes Silveira†

* ITA - Instituto Tecnológico de Aeronáutica
Brazil, SP, São José dos Campos 12228-900
† Federal University of São Paulo (UNIFESP)
Brazil, SP, São José dos Campos 12231-280

Abstract—Pointcut descriptors (PCDs) are an important part of aspect orientation, they describe where a certain behavior will be woven into, it’s correct accuracy is an important issue to be addressed by tests. Previous works on the subject lack on providing a way to independently verify PCDs accuracy, making the test cases less maintainable and more dependent on base code. This paper introduces the JQuati tool, which allows the elimination of the base class dependency, simulating execution contexts and offering a metadata-based syntax which provides a human readable test code for more maintainability and evolutive test development.

Keywords: Aspect Orientation; AOP; Testing; Verification; Validation; Quality; Pointcut Descriptors;

I. INTRODUCTION

As with new features, aspect orientation programming (AOP) has brought to software engineering new complexities. AO works basically by identifying points in the execution of a system, known as join points, by using expressions, called pointcut descriptors, and there inserting additional behavior, known as advice.

On the testing field, aspect orientation has introduced a new kind of testing, not previously seen in other programming paradigms, one aimed to verify whether these pieces of behavior are being woven into the correct join points, a process also known as pointcut descriptor testing. An advice that is executed when not expected can cause failures in the entire system. Previous works on this subject (in Section VI) have shown it as a complicated concern to software testing, lacking of a way to adapt this test to practical and evolutive development cycles.

This paper describes an approach aimed to provide a practical process of testing that includes the test of pointcut descriptors using the JQuati tool. JQuati provides a base class independent, metadata-based testing process, which will help writing human-readable testing code, allowing the use of techniques, such as test-first programming, on aspect orientation.

This paper is divided in 6 Sections: Section II describes other studies on testing in aspect orientation along with the main problem this work aims to address, the testing of PCDs; on Section III, the architecture along with the main components of the solution are detailed; Section IV describes how to use JQuati to develop test cases; in Section V an example is presented providing evidence of how the new testing process will help developer identify faults in pointcut descriptors; Section VI contains some of other existing approaches for this issue; and finally the conclusions along with proposed future works are shown on Section VII.

II. TESTING ON ASPECT ORIENTATION

Testing in aspect oriented languages, can be divided into two steps: verify the correctness of the advice behavior and verify the correctness in the identification of the join points, which means the accuracy of the pointcut descriptors. The JQuati tool focuses on testing such pointcut descriptors for AspectJ[1], which is a aspect-oriented extension to the Java[2] programming language.

Incorrect pointcut descriptors may cause an advice to be woven into unintended join points, which may cause major failures to the system, if that advice tries to access a variable not available or changes unexpectedly the state of a class for instance.

Pointcut descriptor testing is a new issue introduced by aspect orientation. Lemos et al[3] described in their work a fault model, which lists three kinds of errors that might occur when testing pointcut descriptors: (1) too weak pointcut descriptor - pointcut descriptor matches less join points than intended; (2) too strong pointcut descriptor - pointcut descriptor matches more join points than intended; and (3) too weak and too strong pointcut descriptor - when both conditions above occur at the same time.

In AspectJ, some problems were identified which prevent pointcut descriptor of being properly tested, as the dependency on the execution context and the difficulty to intercept when a certain pointcut descriptor has matched a join point. JQuati was developed in order to address these issues while offering an intuitive and simple syntax for development of test cases which verify pointcut descriptor accuracy.

In the topic of testing advice behavior there are several approaches, which can be divided into data flow-based, machine state-based, petri nets-based, and UML model-based.
Data flow-based techniques focus on identifying faults on the use of data throughout the system. This technique was used initially by Zhao [4] and then by Lemos et al [5]. In the first work developed by Zhao, he proposed three levels of testing: intra-module, inter-module and intra-aspect or intra-class [4]. The latter, by [5], a model for representation of data flow of integrated models is proposed along with a criteria of data flow testing based on artifacts generated by AspectJ 1.2 programs.

On techniques based on machine state models, FREE (Flattened Regular Expressions) [6] models are extended to aspect oriented state models, in such a way to specify both classes from the main concerns as aspects from the crosscutting concerns [7]. The first work to present this kind of approach was from Xu, Xu and Nygard [7], presenting an extension of the FREE UML models to aspect orientation. Later, Xu and Xu [8] apply the state models to the behavior of aspects, which are seen as incremental changes to their base classes. Another paper from the same authors was also published on the same technique, focused on integration aspects [9].

Using Petri Net models on testing offers a formal model for developing tests. As with the previously mentioned techniques, it requires careful and strict design of the models. According to Xu and He [10], this kind of approach could narrow the distance between requirements analysis and the subsequent development activities.

UML models-based techniques focus on extracting test cases from aspect oriented models, class diagrams, aspect diagrams and sequence diagrams. Xu and Xu proposed a way for making use of this technique on testing in aspect orientation [11].

Several tools were developed to address the issues on testing aspect orientation, both pointcut descriptor testing and advice behavior testing. Some of these are: aUnit [12], AJTE [13], KTest-METEORA [14], Jabuti/AJ [15], Apte [16], and AdviceTracer[17].

JQuati was designed for PCD testing, but in a way to allow the developer to use other tools and testing approaches along with it. It’s metadata-based syntax provides flexibility by not interfering in the code from the test cases.

### III. JQUATI ARCHITECTURE

JQuati has three main functionalities, which are advice inspection, execution context creation, and expectation management. The components that provide these functionalities are: i) the JQuati runner; ii) the advice inspector; iii) the execution context creator; and iv) the ClassMock framework. Figure 1, illustrates JQuati’s architecture as well as its main components and its functionalities.

The JQuati core runner component orchestrates the use of the other components. It must be set as the runner in the JUnit [18] test class. It reads the metadata on the test class, creating then the execution context, setting the expectations to the advice inspector and after the execution of the test cases, calls the Advice Inspector to verify whether the expectations were met.

Whenever a test case is initialized, the JQuati core runner sets the advice names expected and not expected to be run into the Advice Inspector, which saves this information into different lists. Later, by using a pointcut descriptor meant to match every advice execution in the system, once a advice is executed, it captures the advice name by reading it’s AdviceName annotation, which is then saved into a new list. After the test case execution, the Advice Inspector compares the lists, ensuring that none of the non intended advices were executed and that all of the expected advices were run.

The Execution Context component utilizes the ClassMock framework [19] to generate classes at runtime. The ClassMock framework works by bytecode manipulation with ASM [20]. The Execution Context provides an interface for execution of methods in these new created classes through the Execution Context Element, which is defined by the ExecutionContxtElement annotation.

Although not mentioned as one of its components, the JUnit testing framework has a significant role in the testing process. It invokes all tests and the JQuati core runner callbacks.

JQuati has currently some limitations. One of them resides on not being able to simulate the execution context for PCDs that do not match reflection invocations, such as the call function on AspectJ. The execution context element invokes methods in the runtime created classes by using reflection, thus if a PCD does not match reflection invocations, the execution context element won’t be able to simulate the joint points matched by this PCD.

### IV. USING JQUATI

The JQuati syntax is based on metadata, which means that all the information it needs to execute the tests lies in Annotations. The annotations are used for creation of the execution context, defining test expectations and the execution context element.

The annotation used for creation of the execution context is ExecutionContextCreation, it is applicable to the testing class and receives the parameters ClassNames and MethodNames, which defines the classes names and the methods they will have, these classes will be available for use through the execution context element.

The Annotations available for defining the expectations are the MustExecute and the MustNotExecute Annotation, applicable to testing methods, which must also be annotated with the Test Annotations from JUnit framework. These annotations can be used together to define expectations in the same test method and both receive advice names as parameters. When using a MustExecute Annotation, all calls to the execution context element will have to cause the
advice names received by this Annotation to be invoked, otherwise the test will fail.

The `ExecutionContextElement` Annotation defines which will be the element used for calling methods on the runtime created classes. It is applicable to any `ExecutionContextElement` instance variable in the testing class. The join points to be matched by the tested PCDs are simulated by calling methods on this execution context element. It allows for executing a method on every generated class or on each of them individually.

JQuati verifies faults in PCDs, however, its syntax based on metadata allows it to be used along with other kinds of assertions on the testing methods. One could use it along with mocking tools or use it to also verify the advice behavior.

V. ILLUSTRATIVE EXAMPLE

The concept of pointcut testing is illustrated in this chapter by a simple example: a shooting game. In this game, the player initially chooses the weapon to use and then tries to shoot a certain target with it. According to the weapon used, an animation or sound might be played, therefore detecting when a weapon is fired is an important part of the game requirements.

```java
@ExecutionContextCreation(
    methods=[
        "void shoot(Integer, Integer)"
    ],
    classNames=["Gun", "Pistol", "RocketLauncher"]
)
@RunWith('JQuati', class)
public class SoundEffectsTriggerBehavior {
    @ExecutionContextElement
    ClassExecutionContext cec;
    @Before
    public void setup() {
        cec.instantiateClasses();
    }
    @Test
    @MustExecute({"shootingSound"})
    public void shouldCauseAShootingSound() throws ClassExecutionException {
        cec.executeMethodOnAll("shoot", 0, 0);
    }
}
```

Listing 1. Testing triggering of sound effects with JQuati

The test described in Listing 1 was designed to verify the crosscutting behavior of shooting detection, ensuring that the correct advice is being woven into the correct join points.

Executing this test the JQuati tool warns the developer with a fault in the pointcut strength for too weak pointcut descriptor. As there are no pointcuts implemented yet, no join point is matched and hence the pointcut descriptor is accused to be too weak, which means it is not matching all join points intended.

```java
public aspect SoundEffectsTrigger {
    @AdviceName("shootingSound")
    before() : execution(* . shoot(..)) {
        // make sound effect
    }
}
```

Listing 2. SoundEffectsTrigger class code to pass the test

Listing 2 has some code of the implementation, the pointcut descriptor associated with `shootingSound` advice should match the intended join points in order to pass the test. The join points are expressed by the following form: "Class.Method(ParTypes,...)".

In Table 1, requirements for the shooting game were described and from these, intended and not intended join points were extracted helping in the development of the test cases. The test cases should reflect the intended crosscutting behavior.
Table I

<table>
<thead>
<tr>
<th>requirement</th>
<th>intended joinpoints</th>
<th>not intended joinpoints</th>
<th>test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>a shooting sound should be triggered when any weapon is shot</td>
<td>Gun shoot(0, 0); RocketLauncher shoot(0, 0); Pistol shoot(0, 0)</td>
<td>-</td>
<td>shouldCauseAElosionSound</td>
</tr>
<tr>
<td>Only when a Rocket Launcher is shot, an explosion sound should be triggered</td>
<td>RocketLauncher shoot(0, 0)</td>
<td>Gun shoot(0, 0); Pistol shoot(0, 0)</td>
<td>shouldNotCauseAElosionSound</td>
</tr>
<tr>
<td>a different sound should be triggered when a Pistol shoots out of the limits</td>
<td>-</td>
<td>Gun shoot(100, 100); RocketLauncher shoot(100, 100); Pistol shoot(100, 100)</td>
<td>shouldIdentifyPistolShootingOutOfTheLimits</td>
</tr>
</tbody>
</table>

Table II shows mutations to pointcuts along with the corresponding tests executed with JQuati. These mutations and the subsequent verification with the test cases are intended to help the developer better define the pointcut descriptors that would best address the requirements. Each time a PCD matches more join points than expected in any of the test cases, JQuati warns for a too strong PCD. The same occurs when a PCD neglects a join point, resulting in a too weak PCD warn by JQuati.

The code depicted by Listing 3 describes the test codes from Table II developed with JQuati. The resulting code for the aspect SoundEffectsTrigger can be seen in the Listing 4.

```java
@Test
@MustExecute({"explosionSound"})
public void shouldCauseAElosionSound() throws ClassExecutionException {
    cce.executeMethodOnAll("RocketLauncher", "shoot", -100, -100);
    cce.executeMethodOnAll("RocketLauncher", "shoot", 100, 100);
    cce.executeMethodOnAll("RocketLauncher", "shoot", -100, -100);
}
```

Listing 3. New Test cases in the test class.

```java
public aspect SoundEffectsTrigger {
    @AdviceName("shootingSound")
    before() : execution(* .. shoot(..)) {
        // make sound effect
    }

    @AdviceName("shootingSound")
    before() : execution(* .. shoot(Integer, Integer)) {
        // shooting Sound
    }

    @AdviceName("explosionSound")
    before() : execution(* RocketLauncher..shoot(..)) {
        // explosion Sound
    }

    @AdviceName("pistolShootingOutOfTheLimits")
    before(Integer x, Integer y) : execution(* Pistol..shoot(Integer, Integer)) && args(x,y)
        & if (((x <= 100) || (x >= 100)) || ((y <= 100) || (y >= 100))) {
            // pistol Shooting Out Of The Limits
        }
}
```

Listing 4. Pointcut expressions passing the tests.

VI. RELATED WORKS

The works most related to our approach are the tools AdviceTracer[17], the AJTE[13], APTE[16]. The AdviceTracer[17] tool is very similar to the advice inspector component in JQuati. With it, one can define and verify expectations regarding the execution of certain advices. Although being a useful tool, it does not provide a way to
Table II

<table>
<thead>
<tr>
<th>advice name</th>
<th>candidate expression</th>
<th>test case</th>
<th>precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>shootingSound</td>
<td>execution(&quot;*.shoot()&quot;)</td>
<td>shouldCauseAShootingSound</td>
<td>pass</td>
</tr>
<tr>
<td>explosionSound</td>
<td>execution(&quot;*.shoot(Integer, Integer)&quot;)</td>
<td>shouldCauseAShootingSound</td>
<td>pass</td>
</tr>
<tr>
<td>expansionSound</td>
<td>execution(&quot;*.shoot()&quot;)</td>
<td>shouldCauseAShootingSound</td>
<td>didn't pass: too weak PCD</td>
</tr>
<tr>
<td>pistolShootingOutOfTheLimits</td>
<td>execution(&quot;*.shoot()&quot;)</td>
<td>shouldIdentifyPistolShootingOutOfLimits</td>
<td>pass</td>
</tr>
<tr>
<td>pistolShootingOutOfTheLimits</td>
<td>execution(&quot;*.shoot(Integer, Integer)&quot;)</td>
<td>shouldIdentifyPistolShootingOutOfLimits</td>
<td>pass</td>
</tr>
<tr>
<td>pistolShootingOutOfTheLimits</td>
<td>execution(&quot;*.shoot(Integer, Integer)&quot;)</td>
<td>shouldIdentifyPistolShootingOutOfLimits</td>
<td>didn't pass: too strong PCD</td>
</tr>
</tbody>
</table>

simulate the joint points to be matched by a specific PCD, forcing the developer to write stub classes manually. The more complex a PCD is, the more stub classes the developer would have to build in order to precisely verify that a PCD is matching only the intended joint points, a laborious and error prone task that affects test code maintainability.

AJTE [13] focus on providing a way to develop test cases without weaving, offering a way to test aspects independently from the execution context. When using this tool, classes representing the actual pointcuts are instantiated and receive expressions that should be exactly the same as those from the real aspects, what turns it very dependent on changes that might occur to the actual code. This could cause the test code to become outdated without showing any errors.

The APTE[16] tool aims for automated pointcut testing based on AJTE. This tool comprises a feature to compare different pointcuts. Although also focused on pointcut descriptor testing, JQuati was not built for automated testing, but to offer an easier way for developers to write maintainable test code evolutoively to match ever changing requirements.

VII. CONCLUSION

Aspect orientation has brought new challenges and new open issues for the before existing testing techniques. Previously developed tools and approaches do not offer a way to code base class independent testing of PCDs and lack on providing an understandable syntax for PCD testing, making the development of PCD testing a hard and tiresome task.

As to fill these lacking points, using a metadata-based syntax, JQuati simulates the execution environment, forging the joint points to be matched by the tested pointcuts at runtime and verifies for correctness of pointcut precision by interpreting advice execution. According to the example demonstrated on Section V, JQuati can successfully identify too weak and too strong PCDs, in such a way to allow evolutive development of PCDs and its tests.

In future developments, JQuati should be improved for verification of other kinds of faults that may occur in aspect orientation, as those ones caused by the subsequent execution of two or more advices. Moreover, it should be improved to overcome all of its currently limitations as mentioned in section III. Other additional features that might be added are the diagnosis of a failure using algorithms for comparing pointcut candidates. Additional studies should also be done regarding the use of JQuati in industry and adapting it to mutant testing techniques.

REFERENCES


