The Annotated Test Step Pattern

MARCUS FLORIANO, Aeronautics Institute of Technology
DEBORA CHAMA, Aeronautics Institute of Technology
EDUARDO GUERRA, Aeronautics Institute of Technology
FABIO SILVEIRA, Federal University of Sao Paulo

In the development of automated tests, there is an increase in complexity when the initialization or assertion are related to an external resource. This occurs when the behavior of a class causes a change in the environment where it is running, not only in the application state. Usually these issues are addressed and implemented in the test classes, which make it difficult for their reusage across projects and even into other test classes, generating code duplication and causing loss of productivity when coding tests. This paper presents a pattern that suggests a solution to simplify the initialization and assertion through tests metadata classes. This solution allows each method to has specific assertions and initializations, isolation the solutions out of the test classes, allowing the reusage by other test classes.

Categories and Subject Descriptors: D.1.5 [**Programming Techniques**] Object-oriented Programming; D.2.11 [**Software Architectures**] Patterns

General Terms: Initialization and Assertion

Additional Key Words and Phrases: Automated Tests, Metadata, External Resource

ACM Reference Format:

Floriano, M. and Chama, D. and Guerra, E. and Silveira, F. 2011.ANNOTATED TEST STEP . jn 2, 3, Article 1 (October 2011), 12 pages.

1. INTRODUCTION

There are some techniques that allows the interception of a method to execute additional functionality, such as aspects [Kiczales et al. 1997], dynamic proxies [Forman and Forman 2005] and interceptors [JSR299 2009]. These functionalities usually crosscuts the method functionality and are transparent for the method implementation. Since this crosscutting module are often reused in different classes, it should execute the same functionality. However, sometimes variations of the same crosscuting behavior should be considered for each method, which lead to a question: How to differentiate and configure for each method the execution of a transparent crosscutting behavior?

The CROSSCUTTING METADATA CONFIGURATION [Guerra et al. 2010] proposes the usage of additional metadata to differentiate the behavior of software components that add a crosscutting behavior transparently to an application class. One of the positive consequences is that the same proxy add a custom behavior in each method invoked keeping the decoupling with the application class. However if metadata has a verbose format, it can be more easy to invoke utility methods than to create the declarative configurations.

Author's address: Marcus Floriano, Rua Jose Gomes Faria, 163, Sao Paulo, SP, Brazil, 03819-170; email: marcus.floriano@gmail.com; Debora Chama, Rua Nilza Medeiros Martins, 200, bl5, ap 83, Sao Paulo, SP, Brazil, 05628-010; email: deborachama@gmail.com; Eduardo Guerra, Aeronautical Institute of Technology, Praca Marechal Eduardo Gomes, 50, Vl Acacias, SJ Campos, SP, Brazil; email: guerraem@gmail.com; Fabio Silveira, Science and Technology Institute, Rua Talim, 330, Vila Nair, SJ Campos, SP, Brazil; email:fsilveira@unifesp.br

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission. A preliminary version of this paper was presented in a writers' workshop at the 17th Conference on Pattern Languages of Programs (PLoP). PLoP'11, October 21-23, Portland, OR, USA. Copyright 2011 is held by the author(s). ACM 978-1-4503-0107-7

Automated tests can contain crosscutting functionality, specially when it needs to deal with external resources. In this context, this paper defines external resource as a resource that should be accessed by the test for initialization, assertion or finalization independently from the tested class. Examples of external resources are databases, files and remote services. Based on this definition, a mock object [Freeman et al. 2004] can not be considered an external resource when it should be injected in the tested class. In this context, the crosscutting functionality should vary according to the test scenario or to the expected behavior.

The goal of this paper is to present a pattern that specialize CROSSCUTTING METADATA CONFIGURATION in the context of test automation. This pattern could be included in the pattern collection presented in "Architectural Patterns for Metadata-based Frameworks Usage" [Guerra et al. 2010] considering their usage for the creation of test automation frameworks. It also could be classified as a test pattern and included in the pattern language presented by [Meszaros 2007], as an alternative to decouple and reuse assertion, initialization and finalization logic in test automation code. The pattern ANNOTATED TEST STEP presented in this paper can be specialized into the following three more specific patterns: ANNOTATED SETUP, ANNOTATED ASSERTION and ANNOTATED TEARDOWN. Despite these patterns can provide a better terminology for the solutions, they are not in the scope of this paper.

The target audience for this paper are software developers concerned about the test code quality. The patterns apply for the development of test automation frameworks or when an application needs a large set of tests with the same kind of crosscutting concern. Software architects can use these patterns to facilitate the test creation for architectural components which needs to access external resources.

2. PATTERN: ANNOTATED TEST STEP

2.1 Context

One issue that complicates the creation of automated tests is when the test class should access external resources. Despite the functionality that access the external resource can be isolated and simulated using mock objects [Freeman et al. 2004], at some point it will be necessary to actually test this interaction. That make necessary to initialize and finalize the external resources respectively before and after each test method. For tested classes whose effect is to change an external resource, it is also necessary to assert if its final state is the expected after the test. The access to the same type of external resource usually happens in more than one test class in the same application, so it is desirable to avoid the duplication of this code.

For the test method to be able to access the necessary external resources it usually needs to use classes from another API. This undesirable dependence can make the test code harder to understand, specially when the developer does not know how this other API works. For instance, that can make hard to understand what is the initial test scenario or what is being verified after the tested class execution. This lack of readability can prejudice the test code maintenance and even prevent the usage of the test code as production code documentation.

Despite the access to an external resource can be done independently from the tested class, the test functionality that needs to access it usually is different for each test method. That happens because each one usually needs a different scenario and to verify a distinct effect.

2.2 Problem

How to decouple logic associated to external resources in automated tests, allowing its reuse by more than one test class?

2.3 Motivating Example

To motivate the usage of the proposed pattern, it is used an example of a test class which needs to access a database to setup initial data and to verify its state. For persistence layer testing in an application, it is necessary to perform data loading, removing, updating and verification of such data. DBUnit [Laflamme et al. 2010] is an

example of a test automation framework that can be used to simplify this kind of test. Listing 1 presents a test class using DBUnit classes for the persistence layer of a car rental company application.

It can be observed that several points of the code has explicit reference to the DBUnit classes, which can make this test hard to understand for developers that do not know its API. In the `configureDatabaseConection()` method the integration with the test database is initialized. In addition, in the testing methods `shouldAddCar()` and `shouldRemoveCar()` is necessary to understand the API methods to create the set of expected data and perform the comparisons. As data to setup or to assert the content of database tables is verbose, it is defined in external XML files. An example of an XML file consumed by DBUnit in the example is presented in Listing 2.

LISTING 1: Test class CarDAOTest using DBUnit classes.

```
public class CarDAOTest {

    private CarDAO carDAO = new CarDAO();
    private static JdbcDatabaseTester jdt;

    @BeforeClass
    public static void configureDatabaseConection() throws ClassNotFoundException {
        jdt = new JdbcDatabaseTester("org.hsqldb.jdbcDriver", "jdbc:hsqldb:file:./db/rentacar-db", "sa", "");
    }

    @After
    public void clearDatabaseTable() throws Throwable {
        jdt.onTearDown();
    }

    @Test
    public void shouldAddCar() throws DataSetException, SQLException, Exception{
        loadData("/new-cars.dbunit.xml");
        Car car = new Car("Chevrolet Cobalt", "Compact", true);
        carDAO.add(car);
        ITable actualTable = jdt.getConnection().createDataSet().getTable("tb_car");
        IDataSet expectedDataSet = new FlatXmlDataSetBuilder().build(new File("src-test/car.add.dbunit.xml"));
        ITable expectedTable = expectedDataSet.getTable("tb_car");
        Assertion.assertEqualsIgnoreCols(expectedTable, actualTable, new String[]{"ID"} );
        Assert.assertEquals(3, actualTable.getRowCount());
    }

    @Test
    public void shouldRemoveCar() throws DataSetException, SQLException, Exception {
        loadData("/car.dbunit.xml");
        this.carDAO.remove(1);
        ITable actualTable = jdt.getConnection().createDataSet().getTable("tb_car");
        IDataSet expectedDataSet = new FlatXmlDataSetBuilder().build(new File("src-test/car.remove.dbunit.xml"));
        ITable expectedTable = expectedDataSet.getTable("tb_car");
        Assertion.assertEqualsIgnoreCols(expectedTable, actualTable, new String[]{"ID"} );
    }

    public void loadData(String dataFile) throws Exception {
        DataFileLoader loader = new FlatXmlDataFileLoader();
        IDataSet dataSet = loader.load(dataFile);
        jdt.setDataSet(dataSet);
        jdt.onSetup();
    }
}
```

One of the problems of this test code is the coupling with the DBUnit API, which is used to access the database external resource. This dependence makes the test code more verbose and dependent of an API that is not from the tested class. Despite some functionality could be extracted to more specific utility methods, they can be hard to be reused in other similar test classes. On the other hand the utility methods can also be more general and became a simple adapter of the DBUnit API, creating the same problem but with a different API. The data definition in external files reduce the readability due its indirection, however to include this data in the method's body can

have an even worst effect. The problem in this example is: how make the test class decoupled from the DBUnit API making this logic easy to be reused by other test classes?

LISTING 2: XML file consumed by DBUnit.

```
<!DOCTYPE dataset SYSTEM "rentacar.dbunit.dtd">
<dataset>
        <TB_CAR ID="1" NAME="Ford Escape" TYPE="Compact SUV" CANCELED="false" />
        <TB_CAR ID="2" NAME="Dodge Grand Caravan" TYPE="Minivan" CANCELED="false"/>
        <TB_CAR ID="3" NAME="Chevrolet Cobalt" TYPE="Compact" CANCELED="true"/>
</dataset>
```

2.4 Forces

—Test that require an external resources have a dependency, either directly or through an intermediate API.

—A generic API is more reusable but harder to use; a domain-specific API is easier to use but less broadly reusable.

—A fixture **setUp** method shared by all test methods must satisfy all the tests

2.5 Solution

ANNOTATED TEST STEP use annotations to configure a test step which deals with an external resource. This annotations are processed by a metadata processor which intercepts the invocation of test methods and test life-cycle methods. This processor can execute a functionality before or after the intercepted method. The annotations can indicate that an initialization should be done on the external resource before test, that an verification should be done on the external resource state after test or that an finalization should be done to clean the external resource for the next tests. A premiss of this solution is that the external resource can be accessed by the processor independently of the test class and the tested class, and consequently this logic can be executed separately.

2.6 Resulting Context

The test annotations can be reused in more than one test class that needs to access the external resource. The annotation attributes can parametrize the annotation processor behavior, allowing the usage of the same annotation for different test scenarios.

Since an annotation only configures metadata and do not add behavior, the coupling with the test class is only semantic [Costa Neto et al. 2007]. This kind of dependence is weaker than a method invocation, for instance. Changing the annotation processor, it is possible to easily change the implementation of how the external resource is accessed without changing the test class. This decoupling makes the test class independent of method invocations from APIs that access the external resource. Consequently, the test methods body can depend only on tested class API.

(+). The access to the external resources are decoupled from the test class, but isolated in classes that handle the annotations.

(+). It is possible to add a configuration setting in each specific test method in addition to the shared settings through the use of methods **setup** and **teardown** of the test class.

(+). Initializations, finalizations and assertions related to external resources that are repeated in various test classes can be configured via annotations and reused in different contexts.

(+). Test cases become simpler, making it easier to write and understand, since the use of annotations configure in a declarative way the desired effects.

(+). The configuration and verification are isolated in annotations processors, and the maintenance of the external resource integration is easier since this behavior is isolated.

(+). The behavior to interact to the external resource is isolated in the Metadata Processor, and consequently the test programmers does not need to know the API to interact with it.

(-). Depending on how the annotation was built, the use of annotation restricts the parametrization change at the time of the tests, being restricted only to the settings provided by it.

(-). The usage of annotation is more complex than utility methods and only worth when it can be reused in more than one test class.

(-). This solution makes it more difficult to debug the test cases when problems occur, since the annotations are being processed externally to the test.

2.7 Solution Example

Motivational Example section presented a test of the persistence layer using the DBUnit API. Listing 3 presents these tests applying the pattern ANNOTATED TEST STEP . DBUnit API is no longer used explicitly. Its usage was encapsulated through annotations, and it is no longer necessary to know the the API methods to understand the test. The declarative nature of the annotations transform the invocation of method in configurations that are more easy to understand.

LISTING 3: Test Class CarDAOAnnotationTest using annotations that encapsulate the functionality of DBUnit.

```
@DatabaseConfiguration(driver="org.hsqldb.jdbcDriver", url="jdbc:hsqldb:file:./db/rentacar-db", user="sa", password="")
@RunWith(MakeATestRunner.class)
public class CarDAOAnnotationTest {
    private CarDAO carDAO = new CarDAO();

    @Before
    @CleanDatabaseTable("tb_car")
    public void beforeEachTest() throws SQLException, Exception {    }

    @After
    @CleanDatabaseTable("tb_car")
    public void afterEachTest() throws DataSetException, SQLException, Exception {    }

    @Test
    @GivenDbTableContains(table = "tb_car",
                columns = { "id", "name", "type", "canceled" },
                rows = {"1;Ford Escape;Compact SUV;false",
                        "2;Dodge Grand Caravan;Minivan;false" }
    )
    @DbTableShouldContainOnly(table = "tb_car",
                        columns = { "name", "type", "canceled" },
                        ignoreCols = { "id" },
                        expectedData = { "Ford Escape;Compact SUV;false",
                                         "Dodge Grand Caravan;Minivan;false",
                                         "Chevrolet Cobalt;Compact;true" }    )
    public void whenIAddACar() throws DataSetException, SQLException, Exception {
        Car car = new Car("Chevrolet Cobalt", "Compact", true);
        carDAO.add(car);
    }

    @Test
    @GivenDbTableContains(table = "tb_car",
                columns = { "id", "name", "type", "canceled" },
                rows = {"1;Chevrolet Cobalt;Compact;true",
                        "2;Dodge Grand Caravan;Minivan;false" }
    )
    @DbTableShouldContainOnly(table = "tb_car",
                        columns = { "name", "type", "canceled" },
                        ignoreCols = { "id" },
                        expectedData = { "Dodge Grand Caravan;Minivan;false" }    )
    public void whenIRemoveACar() {
        this.carDAO.remove(1);
    }
}
```

In Listing 3, the configuration of the database is indicated by an annotation called `@DatabaseConfiguration`. In the method `shouldAddCar()` the database setup with the desired data is indicated using another annotation called `@SetupDatabase`, which configures the initial data that should be on the database. Verifications are performed by the processing of specific verification annotations in the end of the test methods. Examples are the `@AssertTableRowCount`, which checks the total of records in the table after running the test, and `@AssertDatabaseContent`, which checks if the content of the table indicated by the parameter table corresponds the data indicated by the parameter expectedData. The `@CleanDatabaseTable` annotation is used in the `afterEachTest()` method to remove all the lines in the car database table after each test.

In this example the framework MakeATest [Floriano et al. 2011] is used to support the processing of annotations, simplifying the task shown in the Sample Code section. The test runner used (indicated by the annotation `@RunWith(MakeATestRunner.class)`) delegates to the framework the identification and process of these configuration and verification annotations.

It may be observed that there is no longer a dependency on DBUnit API. This approach also makes possible to replace the implementation of the annotation processor to integrate with another API, without affecting the test code. The data used for initialization and verification in the tests are no longer in the XML files, reducing the indirection and without polluting the method body with a lot of method calls. Due the fact that annotations are parameterized, it is possible to reuse these annotations in tests that deals with other database tables. As a result it could decrease code duplication in a test suite and increase the productivity in the creation of such tests.

2.8 Known Uses

The JUnit framework [JUnit 2010] [Tahchiev et al. 2010] provides the Excepted Exceptions that aims to determine if the tested code has thrown or not an expected exception. The annotation **@Test(expected= IndexOutOfBoundsException.class)** in a test method indicates that it is expected that an exception of type **IndexOutOfBoundsException** is thrown. The Spring Framework [Spring 2011] [Walls and Breidenbach 2007] also provides a set of annotations called “Common annotations” that can be used in the tests. In this group there is an annotation similar to JUnit Expected Exceptions, called **@ExpectedException**. If it is thrown the exception of the type expected during the test execution, the test is successfully executed. If not, then the test fails.

The JQuati [Santana et al. 2009] is a tool for testing pointcut descriptors in aspect-oriented applications, which simulates and verifies execution contexts and verifies the expectations in advices execution in a more simple and practical way. It contains the annotations **@MustExecute** and **@MustNotExecute** which receive an **Array** with names of advices that are expected to be executed or not, and are assigned in the test cases methods. To use them, it is necessary that the test class be annotated with **@RunWith (JQuati.class)**, which indicates what Runner interferes in the test process, in order to process these annotations.

The MakeATest framework [Floriano and Chama 2011] allows the creation of annotations for assertions related to external resources for the test classes. It enables the creation of test annotations sets to various context domains, supporting the implementation of this pattern.

2.9 Related Patterns

This pattern can be considered as an specialization of CROSSCUTTING METADATA CONFIGURATION [Guerra et al. 2010]. Both of them use the metadata to define crosscutting behavior of application components by using a crosscutting component that interprets the metadata performing the desired behavior. The main difference of ANNOTATED TEST STEP is that it deals specifically with test crosscutting concerns.

ANNOTATED TEST STEP is an alternative to test patterns which allows reuse of the test code in the test initialization, assertion and finalization, such as IMPLICIT SETUP, DELEGATED SETUP, CUSTOM ASSERTION and IMPLICIT TEARDOWN. Considering the problem-pattern cross reference presented in the pattern language proposed by [Meszaros 2007], this pattern can be an alternative for the question “How do we reduce test code duplication?”.

An additional question that could be added in this cross-reference is “How do we decouple the test code from external APIs?”, and would be part of this category the patterns ANNOTATED TEST STEP, DELEGATED SETUP and CUSTOM ASSERTION. The greatest difference between the two other patterns from the one proposed in this paper is that they delegate the functionality to utility methods while this pattern uses annotation configuration with an associated processor.

The usage of the ANNOTATED TEST STEP is more restrict, since it only applies if the test step being encapsulated deals with external resources and needs to be reused in more than one test class. The Listing 4 presents an example which uses DELEGATED SETUP and CUSTOM ASSERTION in the same motivating example presented. The utility methods `loadCars()` and `assertCarTableContains()` encapsulate the use of DBUnit API solving the problem of this test class.

LISTING 4: Using of DELEGATED SETUP and CUSTOM ASSERTION on the motivating example

```
@Test
public void shouldAddCar() throws Exception {
        loadCars("1;Chevrolet Camaro SS;Specialty;true",
                 "2;Dodge Grand Caravan;Minivan;false" ));
        Car car = new Car("Chevrolet Cobalt", "Compact", true);
        carDAO.add(car);
        assertCarTableContains("1;Chevrolet Camaro SS;Specialty;true",
                               "2;Dodge Grand Caravan;Minivan;false",
                               "3:Chevrolet Cobalt;Compact;true"));
}
```

One of the drawbacks of the solution presented in the Listing 4 is that the utility methods are specific to the car table and cannot be reused in others test classes that need to access the database. It is possible to create more general utility methods, however with many information to be configured, the API can became complicated going back to the same initial problem. Due to annotations nature, it separates the configuration of how the external resources should be handled from the method body where the tested class API is being invoked. Listing 5 shows a generic version of listing 4 with the equivalent API to the annotation. Accordingly, the tested class methods are not mixed with methods from other APIs in the test method body. For teams familiar with the annotation notation, this separation can improve test readability and, consequently, its maintenance.

LISTING 5: Generic version

```
@Test
public void shouldAddCar() throws Exception {
        TableConfiguration tbConfig = new TableConfiguration("tb_car");
        tbConfig.setColumns("id" , "name" , "type" , "canceled");
        tbConfig.addLine("1;Chevrolet Camaro SS;Specialty;true");
        tbConfig.addLine("2;Dodge Grand Caravan;Minivan;false");
        tbConfig.configure();
        Car car = new Car("Chevrolet Cobalt", "Compact", true);
        carDAO.add(car);
        tbConfig.assertTableContent("1;Chevrolet Camaro SS;Specialty;true",
                               "2;Dodge Grand Caravan;Minivan;false",
                               "3:Chevrolet Cobalt;Compact;true"));
}
```

With a simple configuration, the use of annotations can be considered very similar to the utility methods for test readability. Listing 6 presents an example when the readability is very similar using both solutions. On the one hand, the greatest drawback of ANNOTATED TEST STEP is the complexity to implement the annotation processor, which can be reduced with a framework like MakeATest [Floriano et al. 2011]. On the other hand, annotations provide a greatest decoupling between the test code and the API that access the external resource. In these cases, the better solution is usually driven by the preference and familiarity of the development team.

LISTING 6: Using of DELEGATED SETUP and CUSTOM ASSERTION on the motivating example

```
//With ANNOTATED TEST STEP
@Test
@CreateFile("testfile.xml")
public void exampleTest() throws Exception {
    //test code
}

//With DELEGATED SETUP
@Test
public void exampleTest() throws Exception {
    createFile("testfile.xml");
    //test code
}
```

2.10 Implementation Notes

To assist in the annotations reading, it can be used the pattern ANNOTATION READER [Guerra et al. 2010], since this pattern proposes the creation of an annotation that can mark the test annotations indicating the class responsible for its reading. The combination of ANNOTATION READER with patterns METADATA PROCESSOR and DELEGATE METADATA READER [Guerra et al. 2009] can be used for reading and processing metadata to allow extension of the metadata schema.

2.10.1 *Structure.* To use this approach it is necessary to have a mechanism for identifying the **Metadata** when the **Test Executor** starts the tests execution. This way, the **Metadata** of the **Test Class** is recovered and processed, and the related functionality executed before or after the test method.

The **External Resources Manager** should be used to encapsulate access to **Test Class** by **Test Executor**. For each execution of the test methods, the **External Resources Manager** invokes when necessary the **Metadata Processor** for initialization of **External Resources** related to the **Target Class**, based on configured **Metadata**. Then, it invokes the test method of the **Test Class**, and if necessary, invokes the **Metadata Processor** to verify the expected modifications on the **External Resources** caused by the execution of the **Target Class**.

The initialization and verification processes related to **External Resources** use the **Metadata** of test method to set the desired behavior to be executed.

The **External Resources Manager**, which controls the execution of test methods intercepting them and calling the initialization and verification of **External Resources**, can be implemented as a dynamic proxy, an aspect or as a part of the test runner.

Figure 1 depicts the pattern solution. When the **Test Executor** starts the tests execution, it is intercepted by a **External Resources Manager**. For each **Test Class**, this manager delegates to the **Metadata Processor** the task of reading the **Metadata** on the test methods. After and before the invocation of the test method, the **Metadata Processor** checks **Metadata** if there is a functionality that must be executed. These **Metadata** refer to **External Resources** that should be initialized before testing the **Target Class** or should checked after the tests execution.

Some **Metadata** must be processed before the test method for the creation of the initial test scenario. Others must be processed after the execution of the test methods in order to assert the expected **External Resources** state caused by execution of the **Target Class** or to clean the changes in the **External Resources** for the execution of other test methods.

After such assertions, **Metadata Processor** can verify if the test failed or if it passed. More than one **Metadata** piece can exist in each method, and all **Metadata** should be processed to ensure that the desired behaviors were achieved.

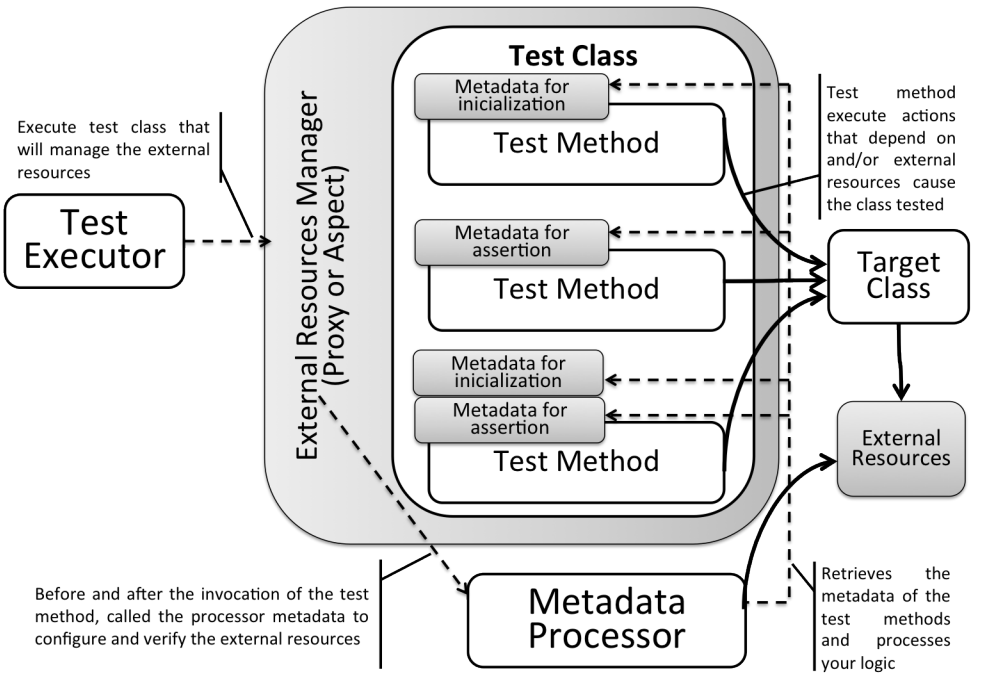

Fig. 1: Graphical representation of communication of participants pattern solution.

2.10.2 *Participants.* The participants of the pattern are:

—**External Resources** are resources that are part of the environment where the application runs that need to be initialized before or checked after tests.

—**Target Class** are classes that perform some action which depends on external resources or changes them.

—**Test Class** contains test methods with metadata for configuring or verifying the external resources of the target class.

—**Test Executor** is the executor that initiates and manage the tests execution.

—**External Resources Manager** manages the execution of actions and check external resources during the tests execution. Before calling the test method itself, it checks the metadata for initialization and, after running the test method, it calls for verification of external resources related to these metadata. Typically implemented as a dynamic proxy or as an aspect (by using aspect-oriented programming).

—**Metadata** contains the information about the test class or the test method refers to initialization or verification of external resources.

—**Metadata Processor** is responsible for retrieving and interpreting the metadata of the test methods and test classes, and execute the functionality related with the metadata.

2.10.3 *Sample Implementation Code.* To present the usage of this pattern it was developed this sample code that checks a **External Resource** after the test execution. The objective is to verify that a given property in a properties file was correctly written.

First of all, it was created the class to be tested called `PropertyFile`, which has the responsibility to record and retrieve values from properties file. Listing 7 shows the interface of such a class. After that, the **Metadata** – represented by the annotation `ValidatePropertyFile` – was created, as shown in listing 8, which is used to annotate the test method of the **Test Class**.

LISTING 7: Target Class `PropertyFile` which is responsible for recording and retrieving properties from a properties files.

```
public class PropertyFile {
    public PropertyFile(String fileName){ ... }
    public void write(String property, String value){ ... }
    public String read(String property){ ... }
}
```

LISTING 8: Metadata: Annotaion `@ValidatePropertyFile` that represents the validation of a property in the properties file.

```
@Target({ ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
public @interface ValidatePropertyFile {
    String file();
    String property();
    String value();
}
```

Listing 9 shows the **Test Class** `ValidatePropertyFileTest`, where the created annotation was used. If the properties file has the expected value for a particular property, the test runs successfully. Otherwise an exception should be thrown and the test would fail.

To process this annotation it was used an aspect as an **External Resources Manager**. Listing 10 displays the aspect `AspectManager`, developed by using AspectJ [Laddad 2009], which indicates that any execution method annotated with `ValidatePropertyFile` should be intercepted.

When the **Test Executor** begins the execution of the **Test Class**, the aspect will manage the processing of annotations. The advice `manager` from `AspectManager` identifies which methods contain the annotation `ValidatePropertyFile` to invoke the **Metadata Processor** (`PropertyFileMetadataProcessor`), which is responsible for interpreting the annotations.

The `PropertyFileMetadataProcessor` presented in listing 11 retrieves from the annotation `ValidatePropertyFile` the values “file”, “property” and “value”, and then verifies if the property is configured in the file with the indicated value. If not, an exception is thrown and the test fails. Otherwise the test will end successfully.

LISTING 9: Test Class: **ValidatePropertyFileTest** to test the annotation `@ValidatePropertyFile`

```
public class ValidatePropertyFileTest {
    private PropertyFile propertyFile;
    @Before
    public void setUp() {
        try {
            this.propertyFile = new PropertyFile("application.properties");
            this.propertyFile.write("property", "propertyValue");
        } catch (Exception e) {
            Assert.fail();
        }
    }

    @Test
    @ValidatePropertyFile(file = "application.properties", property = "property", value = "anotherValue")
    public void verifyProperty() {
    Assert.assertEquals("propertyValue", this.propertyFile.read("property"));
        try {
            this.propertyFile.write("property", "anotherValue");
        } catch (Exception e) {
            Assert.fail();
        }
    }
}
```

LISTING 10: External Resources Manager: `AspectManager`.

```
@Aspect
public class AspectManager {
    @Around("execution(@ValidatePropertyFile * *(..)) && @annotation(validatePropertyFile)")
    public void manager(ProceedingJoinPoint jp, ValidatePropertyFile validatePropertyFile) throws Throwable {
        try {
            jp.proceed();
            PropertyFileMetadataProcessor processor = new PropertyFileMetadataProcessor();
            processor.processAnnotation(validatePropertyFile);
        } catch (Exception e) {
            Exception exception = new Exception("Exception in processor class: " + e);
            exception.setStackTrace(e.getStackTrace());
            throw exception;
        }
    }
}
```

LISTING 11: Metadata Processor: `PropertyFileMetadataProcessor`.

```
public class PropertyFileMetadataProcessor {
    public void processAnnotation(ValidatePropertyFile validatePropertyFile) throws Exception {
        try {
            Properties props = new Properties();
            props.load(new FileInputStream(validatePropertyFile.file()));
            Assert.assertEquals(validatePropertyFile.value(), props.getProperty(validatePropertyFile.property()));
        } catch (FileNotFoundException e) {
            throw new AssertionError(e.getMessage());
        } catch (IOException e) {
            throw new AssertionError(e.getMessage());
        }
    }
}
```

3. CONCLUSION

This paper presented a pattern where the metadata usage helps to decouple logic associated to external resources in automated tests, allowing its reuse by more than one test class. This decoupling makes the test class independent of method invocations from APIs that access the external resource.

However, the way the logic is decoupled must be suitable for the scenario, since the usage of annotation is more complex than the traditional utility methods and only worth when it can be reused in more than one test class.

ACKNOWLEDGMENTS

We would like to thank our shepherd Gerard Meszaros for all the suggestions during the writing of this paper. We also would like to thank for the essential support of FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo) to this research.

REFERENCES

COSTA NETO, A., RIBEIRO, M., DÓSEA, M., BONIFÁCIO, R., AND BORBA, P. 2007. Semantic dependencies and modularity of aspect-oriented software. In *Proceedings...* WORKSHOP ON ASSESSMENT OF CONTEMPORARY MODULARIZATION TECHNIQUES, 2007.

FLORIANO, M. AND CHAMA, D. 2011. Makeatest - core. Available on: http://github.com/marcusfloriano/makeatest-core.

FLORIANO, M., CHAMA, D., GUERRA, E., AND SILVEIRA, F. 2011. Makeatest: Um framework para construção de anotações de validação e inicialização de recursos externos em testes automatizados. 5th Systematic and Automated Software Testing (SAST2011), Sao Paulo, SP, Brazil.

FORMAN, I. AND FORMAN, N. 2005. *Java Reflection in Action*. Manning Publications Co.

FREEMAN, S., PRYCE, N., MACKINNON, T., AND WALNES, J. 2004. Mock roles, not objects. In *Proceedings...* OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, New York, NY, USA, 236–246.

GUERRA, E., FERNANDES, C., AND SILVEIRA, F. 2010. Architectural patterns for metadata-based frameworks usage. In *Proceedings...* 17th Conference on Pattern Languages of Programs (PLoP), Reno, Nevada, USA.

GUERRA, E., MENANES, C., SILVA, J., AND FERNANDES, C. 2010. Idioms for code annotations in the java language. In *Proceedings...* 17th Latin-American Conference on Pattern Languages of Programs (SugarLoafPLoP), Salvador, Bahia, Brasil, 14.

GUERRA, E., SOUZA, J., AND FERNANDES, C. 2009. A pattern language for metadata-based frameworks. 16th Conference on Pattern Languages of Programming, Chicago.

JSR299. 2009. Jsr 299: Contexts and dependency injection for the javatm ee platform. Available on: http://jcp.org/en/jsr/detail?id=299.

JUNIT. 2010. Junit.org resources for test driven development. Available on: http://junit.org/.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C. V., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *Proceedings...* EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 220–242.

LADDAD, R. 2009. *AspectJ in Action, Second Edition*. Manning Publications Co.

LAFLAMME, M., GIACCO, R. L., LEME, F., AND PUGH, E. 2010. Dbunit is a junit extension targeted at database-driven. Available on: http://www.dbunit.org.

MESZAROS, G. 2007. *XUnit test patterns: refactoring test code*. Person Education, Inc.

SANTANA, E. C., TANAKA, S. H., GUERRA, E. M., FERNANDES, C. T., AND SILVEIRA, F. 2009. Towards a practical approach to testing pointcut descriptors with jquati. In *Proceedings...* III Latin American Workshop on Aspect-Oriented Software Development - LA-WASP'2009, Fortaleza.

SPRING. 2011. Spring framework. Available on: http://www.springsource.org.

TAHCHIEV, P., LEME, F., MASSOL, V., AND GREGORY, G. 2010. *JUnit in Action, Second Edition*. Manning Publications Co.

WALLS, C. AND BREIDENBACH, R. 2007. *Spring in Action, Second Edition*. Manning Publications Co.

PLoP'11, October 21-23, Portland, OR, USA. Copyright 2011 is held by the author(s). ACM 978-1-4503-0107-7