Testing Asynchronous Information Systems with ThreadControl: a Case Study

Diego Sousa[1], Ayla Dantas[1], Ewerton Lopes[1]

[1]Departamento de Ciências Exatas – Universidade Federal da Paraíba (UFPB)
Campus IV – Rua da Mangueira, S/N – Companhia de Tecidos Rio Tinto
CEP 58297-000 – Rio Tinto – PB - Brazil

{diego.sousa,ayla,ewerton.lopes}@dce.ufpb.br

Abstract. *This paper presents a case study regarding the use of the ThreadControl Testing Framework to test asynchronous information systems. This framework is intended to help test developers to avoid false positives in the execution of their automatic tests for asynchronous systems. Some of these false positives occur because the tests fail in some executions due to early or late verifications (assertions). In this paper, we have used ThreadControl in the automatic tests of a simple CRM information system that is asynchronous and discuss the main challenges and lessons learned from this use.*

Resumo. *Este artigo apresenta um estudo de caso sobre o uso do arcabouço de testes ThreadControl para testar sistemas de informação assíncronos. Este arcabouço busca ajudar desenvolvedores de testes a evitar falsos positivos na execução de seus testes automáticos para sistemas assíncronos. Alguns desses falsos positivos ocorrem porque os testes falham em algumas de suas execuções devido a verificações (asserções) antecipadas ou tardias. Neste artigo, o ThreadControl foi utilizado nos testes automáticos de um sistema de informação CRM simples e assíncrono. Com base neste uso, este trabalho discute alguns dos principais desafios e lições aprendidas identificadas.*

1. Introduction

Software testing is challenging. When we test a system, we want to stimulate its operations and verify if the results produced by such operations are as expected. Tests, and specially automatic tests are really important for the evolution of systems, specially today, when this evolution should be very fast. Many online information systems we use today are updated very quickly and in order to support such quickness, we need tests that can be rapidly executed to verify if even after some changes, such as the introduction of a new feature or a bug fix, the system is still working as expected (at least for the scenarios covered by the tests).

When a test is failing, it is a sign that something is broken in the system and that a defect should be found. However, when we test asynchronous systems, some false positives may happen, which may cause several problems for the development team. Sometimes the test can fail because the test verification is performed too early, when the results expected by the test have not been produced yet, or too late, when the system is already in another state, after reaching the expected state of its threads. A common cause for this problem is the use of explicit delays (such as `Thread.sleep(timeout)`) or busy waits (explicit delays inside loops) in tests between the stimulation and verification phases, because it is a simpler alternative than synchronization mechanisms

between the test and the System Under Test (SUT), which may be too invasive considering the necessity of changes in the SUT.

Dantas et al. (2008) has proposed a non-invasive alternative to provide such synchronization and avoid late and early assertions: the ThreadControl Testing Framework. This framework is implemented in Java and AspectJ (an Aspect-Oriented extension to Java) [Kiczales et al. 2001].

In this work we discuss how this framework can be used in practice through the description of a case study using a simple Customer Relationship Management (CRM) system. We believe that the description of our use of ThreadControl can be very useful for other information systems test developers in better understanding this framework and how they can avoid false positives in tests of asynchronous information systems. Besides, the main lessons collected from our case study may also give test developers an idea of the challenges and good practices to use the ThreadControl Testing Framework. Our main observation is that using ThreadControl is not always simple, but some good practices can be very helpful, such as correctly understanding the behavior of the system threads and creating auxiliary methods in the tests with the help of SUT developers to specify threads states that are desired at the moment assertions are being performed. Besides, evolving the framework is also challenging due to its use of Aspect-Oriented Programming (AOP) and the Java concurrency API, but such effort is worthwhile as we do not need to change the SUT code in order to get a better control of its threads during tests with AOP.

The remaining of this paper is organized as follows. In Section 2 we present some background information to make the reader familiar with some topics that are discussed. In Section 3 we describe the case study that has been performed and how the ThreadControl Framework has been used to test a simple CRM. In Section 4 we present the main challenges observed and the lessons learned from this experience. In Section 5 we present some related work. Finally, in Section 6 we conclude this paper and point out directions for future research.

2. Background

There are several kinds and classifications of information systems. In this section we briefly present some of the main types of information systems, especially CRMs, as a CRM was chosen for the case study presented in this work. Besides, we present an overview about testing asynchronous systems. Moreover, we describe the ThreadControl Testing Framework and how it can be used.

2.1 Information Systems Types

Information systems are created to support decision makers in deciding which strategies to adopt in an organization. For Stair and Reynolds (2002, pg. 12), an information system is a set of elements and interrelated components that collect (input), manipulate (process) and disseminate (output) data and information and provide feedback mechanisms to achieve an objective.

Due to the existence of different interests, specialties and levels in an organization, there are different types of information systems, serving different needs of the organization. These include: Customer Relationship Management (CRM) specialized in customer relationship; Enterprise Resource Planning (ERP), responsible

for the integration of information in an organization; Management Information Systems (MIS), with the ability to generate information needed for decision making; Transaction Processing Systems (TPS), which perform and record the daily transactions of an organization; among others.

In the case study used in this article, we will use a CRM system. CRMs, according to Fingar et al. (2000), were motivated by the fact that, in some cases, customers that look to a company have a fragmented view of it. Each view presents the characteristics of the sector with which the client is interacting. The role of CRM is exactly to eliminate the separation of client information in areas within an organization. In other words, all sectors should share information about the client, resulting in a central database accessible to all areas.

2.2 Testing Asynchronous Systems

Software testing is an area of software engineering intended to improve the confidence that the software will behave correctly and to check if the product produced meets the specifications required by the customer. Tests are among the approaches for verification and validation (V & V) of systems and they are characterized as a technique in which the software is exercised for possible defects, to increase the confidence that it will behave correctly when in production [Sommerville 2006]. This can be done manually or through automated test tools, which can create a more efficient, and repeatable test environment, improving the quality of the test effort and minimizing the schedule [Dustin et al. 1999].

Sometimes, while automatically testing concurrent systems, the operations being exercised are asynchronous, which makes testing a challenging task as automated testing tools need to deal with local non-determinism and imperfect communication channels [Salas and Krishnan 2009]. Besides, when writing tests for asynchronous code, we will have to deal with the fact that the lack of synchronization between the system and the test may lead to race conditions between the test and the SUT. When the operations being tested are synchronous the tests are much simpler. For instance, after invoking an operation, such as `insertNewProduct(productId, amount)`, we can make the test verify if the product has been inserted without including in the test any delays. With asynchronous operations we do not have such guarantees, which may result in false positives in some executions of the tests. In other words, the test may fail but the defect is not in the SUT but in the test, which can make developers become suspicious of test results or, on the other hand, this may also make them lose a lot of time looking for a bug that does not exist [Dantas et al. 2008]. However, many developers still use in their tests solutions such as explicit delays (`Thread.sleep(timeout)`) or busy waits, which may make the tests fail sometimes or can also make them take a lot more time than necessary if the timeouts chosen are too big in order to avoid the false positives. If we consider that for each verification a large delay should be included, such delays can make the execution of test suites to take hours instead of minutes.

2.3 Testing systems using ThreadControl

Dantas (2010) describes a test framework called ThreadControl and a general approach for testing asynchronous systems, based on monitoring the application threads, called "Thread Control For Tests". This framework offers an API with test primitives to aid the development of automatic tests for multi-threaded systems, avoiding test assertions

that can be performed too early or too late (such as the ones that happen when the tester prefers to use explicit delays or busy waits). The ThreadControl framework is based on the monitoring and control of the system threads during the execution of automatic tests (such as the ones implemented using the JUnit [Massol 2004] framework, for instance). In order to be non-invasive regarding the necessity of changes in the SUT, this framework is implemented using AspectJ [Kickzales et al. 1997].

Initial versions of this framework, which is available as open source at http://code.google.com/p/threadcontrol/, are based on the monitoring of the main operations related to threads in the API (Application Programming Interface) for version 1.4 of the Java programming language and of some of the operations of the concurrency API introduced in version 5 (package java.util.concurrent). This means that multiple systems can benefit from this framework for the preparation of their tests even without the necessity of changing the framework code, which is in Java and AspectJ. In order to use the ThreadControl testing framework in tests, it is necessary to install the AspectJ Development Tools (AJDT) plugin in the IDE being used. Besides that, we must download the ThreadControl code package by accessing http://code.google.com/p/threadcontrol/downloads/list and include this code in the project source code. The framework provides some primitives to be used in the tests.

The general idea of the ThreadControl framework is that the following test primitives (offered by methods from the ThreadControl facade class) are provided to test developers when they deal with asynchronous operations in automatic tests:

- **prepare:** it is used to specify desired system configurations (in terms of thread states) for which the system should wait for before performing assertions. These configurations represent phases in the system execution which correspond to certain threads' states and are passed as a parameter for the prepare operation;
- **waitUntilStateIsReached:** it allows the assertions to be performed at a secure moment, when the state specified through the prepare operation has actually been reached. If a monitored thread tries to disturb the system after this moment, it will be blocked in order to avoid a late assertion. When this operation is called, the test thread waits until the application has achieved the expected state and then performs the assertions;
- **proceed:** this is the operation responsible for making the system proceed its normal execution. Any threads that were not allowed to proceed because assertions were being performed are then released. This way, the test may terminate or continue with other assertions.

In order to illustrate how the ThreadControl is used, we present in the following a code snippet of a JUnit test case for some classes based on the Producer and Consumer example.

```
1 public class AppTest {
2  @Test
3  public void testUsingJustAProducer() {
4   ThreadControltc = newThreadControl();
5   Buffer buffer = newBuffer(Buffer.LIMIT);
6   Producer prod1 = new Producer(buffer);
7   tc.prepare(getSysConfigOfProducerWaiting());
8   new Thread(prod1).start();
9   tc.waitUntilStateIsReached();
```

```
10  assertEquals(Buffer.LIMIT,buffer.size());
11  tc.proceed();
12  producer.stop();
13 } //End of the test method;
14
15 public SystemConfiguration getSysConfigOfProducerWaiting() {
16   ThreadConfiguration config = new ThreadConfiguration(
17     Producer.class.getCanonicalName(),ThreadState.WAITING,
18     ThreadConfiguration.AT_LEAST_ONCE,
19         ThreadConfiguration.ALL_THREADS_TO_BE_IN_STATE);
20  ListOfThreadConfigurations sysConfig=new ListOfThreadConfigurations();
21  sysConfig.addThreadConfiguration(config);
22  return sysConfig;
23 } // End of the method
```

In this example, the `Producer` class is a runnable class that generates messages to be stored in a `Buffer` object from time to time, sleeping between these intervals. If the buffer limit is reached, this `Producer` waits until a consumer object removes an item from the buffer before producing more messages.

The test method above checks if when the Producer runnable reaches the waiting state, the limit of the buffer has been actually reached. In order to do that, an instance of the `ThreadControl` class should be created (line 4) to allow the invocation of the test primitives used to make the test thread wait until the expected state has been reached. Besides, the test should initialize the objects regarding the classes to be tested. In this test, we initialize the classes `Buffer` and `Producer` (lines 5-6).

In the following, we should inform `ThreadControl` about the expected state to be reached before the next assertion is performed. This is done through the invocation of the `tc.prepare` method, which receives as parameter an object from type `SystemConfiguration`. From this moment on, ThreadControl begins monitoring the SUT considering the specified configuration of the system threads, which is, in this case, the state in which all instances of the `Producer` class are in the `WAITING` state and this state has been reached at least once (lines 17-19). After preparing ThreadControl, the test actually starts the Thread that will make the `Producer` start its execution (line 8). Then, the test should verify if the buffer limit is reached when the Producer reaches the `WAITING` state. In order to make the test wait until this state is reached, we insert an invocation to the `waitUntilStateIsReached`() method (line 9). This invocation blocks the test thread until the expected state is reached. Then, the test assertion verifies that the buffer size has reached its limit (line 10).

Finally, after performing test assertions that depend on the achievement of the expected state, the `proceed`() method from ThreadControl is called (line 11) in order to release any thread that could be previously blocked when trying to change its state while assertions were being performed.

The method shown between lines 15-23 is used to create an object from type `SystemConfiguration`, an interface which represents configurations of expected states of interest for the test developers. The object created in this method is an instance of the `ListOfThreadConfigurations` class in which the expected state for threads identified by the `Producer` class name is `WAITING`.

3. Case Study: Using ThreadControl to test an asynchronous CRM

In this section we describe our experience in using ThreadControl to test a simple CRM system implemented using asynchronous operations.

3.1. Overview of the MySimpleCRM system

The system used for the case study presented in this work is a simple electronic system for customer relationship management (CRM), which was called MySimpleCRM. It basically has the following asynchronous operations:

- Management of customers, products and promotions;
- Management of purchases information and promotions;
- Automatically sending emails to clients informing about purchases;
- Automatically sending emails wishing congratulations on the customers birthdays;
- Automatically sending emails about Promotions according to the profile of each customer.

MySimpleCRM is a web system. Its architecture is divided in three layers: User Interface Layer, Services Business Layer and Data Layer. The User Interface Layer has a client application responsible for the interaction of the system with a user using a browser or with another service. The Services Business Layer contains all the code responsible for receiving and processing the data from the user interface layer and can interact with the Data Layer to store any data. Inside the Services Business Layer, we have defined a sub layer called "Logic Layer", which concentrates all the business logic of the application, and an "Asynchronous Layer" responsible for performing the system operations in an asynchronous way using a thread pool. The system logic is accessed through the `Facade` class, which interacts with the Asynchronous Layer.

The MySimpleCRM follows the SOA (Service-Oriented Architecture) architectural approach and its services are offered through generic interfaces provided by REST (Representational State Transfer), which uses HTTP messages (Hypertext Transfer Protocol) for data communication. Using this approach we can get highly decoupled modules, ease of reuse and scalability, among other advantages.

In order to better understand the MySimpleCRM system, we illustrate in Figure 1 some of its main classes, specially the ones used when an `addPurchase` operation is invoked in this system.

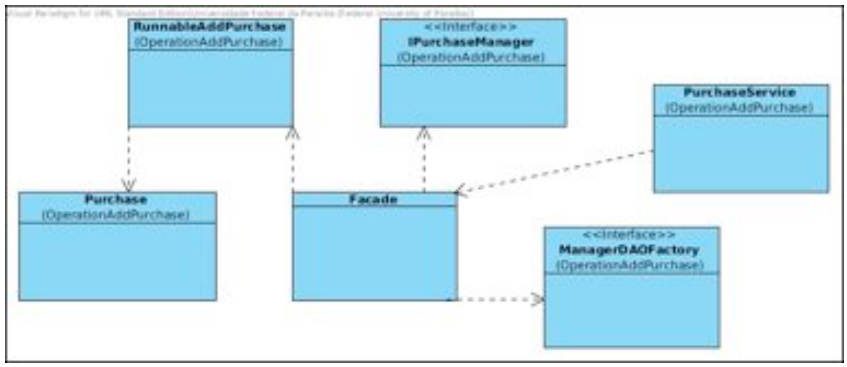

Figure 1: Main classes involved with the addPurchase operation.

The class `PurchaseService` is a class of the SOA layer that is responsible for the communication between the logic layer and the user interface layer. The `Facade` class provides a simplified interface to access the system functionalities. The `Purchase` class is the entity that represents a client purchase. As the MySimpleCRM system uses the DAO (Data Access Object) pattern to separate business rules from the rules database access, it provides interfaces such as the `IPurchaseManager` interface illustrated in Figure 1. This interface, in particular, represents persistence operations regarding the `Purchase` entity. Some examples of the implementations of this interface are the classes `PurchaseDAOJPA` and `PurchaseDAOFILE`, which use JPA (Java Persistence API) and simple files as persistence mechanisms, respectively. The Abstract Factory design pattern [Gamma et al. 1994] is used in the system in order to build these different implementations. For instance, when the persistence mechanism used is JPA, the system uses an instance of the `ManagerDAOFactoryJPA`class, which implements the `ManagerDAOFactory` interface (shown in Figure 1), to build the specific `IPurchaseManager` object.

In order to provide its operations in an asynchronous way, the MySimpleCRM system uses some classes that implement the Java Runnable interface, such as the `RunnableAddPurchase` class, also shown in Figure 1.When this object starts its execution (when its run method is being executed), it uses an `IPurchaseManager` object (such as an instance of the `PurchaseDAOJPA` class) to actually perform the purchase management.

The sequence diagram, illustrated by Figure 2, describes the interaction between the described classes of the MySimpleCRMsystem regarding a product purchase.

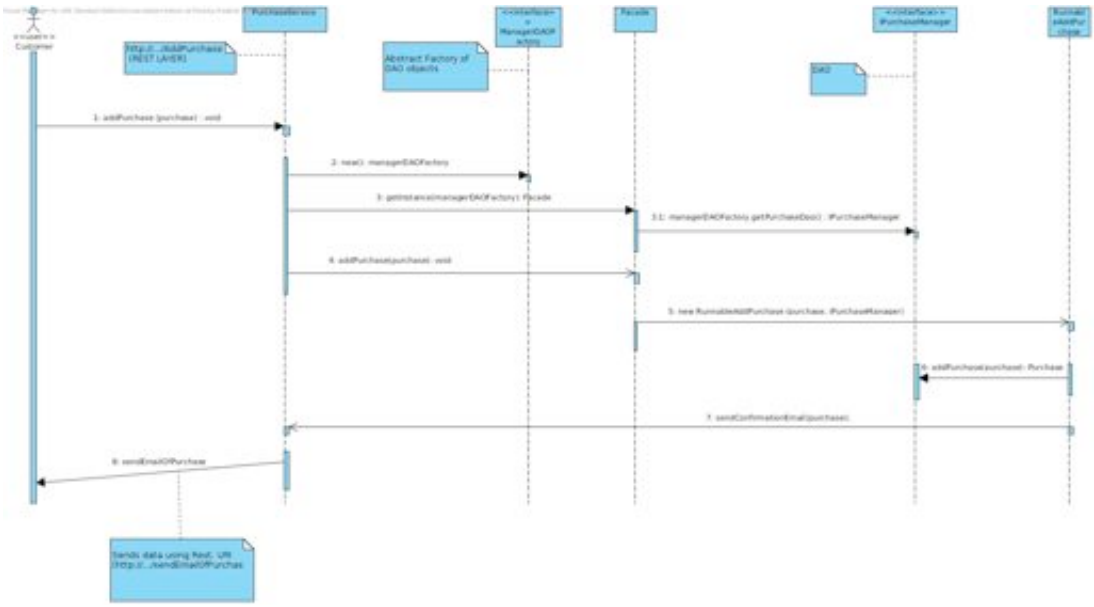

Figure 2: Interaction between some elements of MySimpleCRM when a purchase is performed

3.2. Automatically Testing MyFirstCRM

In order to test a scenario regarding the use of the MySimpleCRM during a product purchase, we can use the JUnit and ThreadControl testing frameworks following the steps bellow:

1. Create one instance of ThreadControl (tc) in order to use the primitives to make the test wait until a certain state before executing some assertions.
2. Create one instance of the MySimpleCRM `Facade` class to be used while exercising the system operations.
3. Register a new Customer called “customer1”.
4. Wait until the threads responsible for adding customers have finished their job.
5. Verify if the registered customer can be found in the system.
6. Register a new Product named “product1”.
7. Wait until the threads responsible for adding products have finished.
8. Verify if the new Product can be found in the system.
9. Register a new Purchase identified by “purchase1” regarding “product1” for the “customer1” client.
10. Wait until the threads responsible for adding purchases have finished.
11. Verify if the Purchase named “purchase1” was correctly included in the system.

As the system operations invoked through the Facade class are asynchronous, the test thread should wait some time before executing the code responsible for the assertions. To avoid false positives by the use of inadequate explicit delays (such as `Thread.sleep` calls) with insufficient timeouts, we have used the ThreadControl testing framework, as illustrated by the commented code snippet shown in the following. This code corresponds to steps 1-8 previously mentioned. The steps 9-11 represent the finalization of the purchase operation. They are not explained here because they are similar to the steps 3-5.

Source Code 1: JUnitTest case regarding the initial steps of a purchase

```
1 @Test
2 public void testAddPurchase() {
3
4  threadControl = new ThreadControl(); //STEP 1
5  facade = Facade.getInstance(new ManagerDAOFactoryJPA()); //STEP 2
6
7  /*------------Starting operations of the customer------------*/
8
9  takerClientList = new LinkedBlockingQueue<Customer>();
10 CustomercustomerAuxOne = null;
11 Customer customer1 = new Customer("Diego", "07278910112", "3422-1010",
12 "diego.sousa@dce.ufpb.br", "S3cr3t", 18, 11, 1988);
13
14 // Add Customer (previously specifying the test waiting condition)
15 threadControl.prepare(getAddCustomerFinishedState(1));
16 facade.addCustomer(customer1); //STEP 3
17 threadControl.waitUntilStateIsReached(); //STEP 4
18 threadControl.proceed();
19 // getting customer by CPF
20 facade.searchCustomerByCpf("07278910112", takerClientList); //STEP 5
21
22 try {
23 customerAuxOne = takerClientList.take(); //this remains synchronous due to
24 } catch (InterruptedException e) {         //the use of a BlockingQueue
25  e.printStackTrace();
26 }
27
28 assertEquals("Diego", customerAuxOne.getName());
29 assertEquals("diego.sousa@dce.ufpb.br", customerAuxOne.getLogin());
```

```
30
31 /*----------------End of customer operations------------------*/
32
33 /*------------Starting operations of the product--------------*/
34
35 takerProductList = new LinkedBlockingQueue<Product>();
36 Product productAuxOne = null;
37 Product product1 = new Product("IPod", 1200.00, 100);
38
39 //Add Product
40 threadControl.prepare(getAddProductFinishedState(1));
41 facade.addProduct(product1); //STEP 6
42 threadControl.waitUntilStateIsReached(); //STEP 7
43 threadControl.proceed();
44
45 // Searching by product name
46 facade.searchProductByName("IPod", takerProductList); //STEP 8
47
48 try {
49  productAuxOne = takerProductList.take();
50 } catch (InterruptedException e) {
51  e.printStackTrace();
52 }
53
54 assertEquals("IPod", productAuxOne.getName());
55 assertTrue(productAuxOne.getPrice() == 1200.0);
56 //...
57/*-----------------End of Product operations------------------*/
```

In order to define the expected state for the system at the moment that assertions are executed, the test developer should build a `SystemConfiguration` object. As the same expected state can be used for several tests, it is interesting to develop methods for creating such objects, as the `getAddCustomerFinishedState` method presented in the following code snippet.

Source Code 2: Auxiliary method to be used by JUnitTest cases that invoke the operation `addCustomer`

```
1 public SystemConfiguration getAddCustomerFinishedState(int timesToBeInState) {
2   ThreadConfiguration config = new ThreadConfiguration(
3     RunnableAddCustomer.class.getCanonicalName(),
4     ThreadState.FINISHED, timesToBeInState);
5   ListOfThreadConfigurations sysConfig = new ListOfThreadConfigurations();
6   sysConfig.addThreadConfiguration(config);
7   return sysConfig;
8 }
```

This method creates an instance of the `ListOfThreadConfigurations` class, which is an implementation of the `SystemConfiguration` interface. The configuration built by this object corresponds to a state in which all instances of the `RunnableAddCustomer` class are in the `FINISHED` state and the number of times runnables from this class have reached this state is `timesToBeInState`. In the test code, the value used for this parameter was "1" (line 15 of source code 1), which means that the test should wait until one instance of the `RunnableAddCustomer` has finished its execution.

4. Lessons learned from using and evolving ThreadControl

After developing some automatic tests for the MySimpleCRM system we have noticed that the current version of the ThreadControl testing framework (version 0.3) was not

completely supporting thread operations regarding thread pools. Therefore, we have tried to extend the framework in order to provide this support, but we have also looked for alternatives to test the system even without these extensions. The main lessons learned from this experience are explained in this section.

Before using ThreadControl to make the tests wait before test assertions, we have used the alternative of using invocations to `Thread.sleep(timeout)`. As expected, sometimes we had executions where the tests failed because the timeout chosen was not enough. For an extended version of the `testAddPurchase()` previously shown we have seen that using a timeout of 500 milliseconds, we did not found any failures in a thousand executions of the same test. However, using a timeout of 250 milliseconds (ms), we had 80 failures in a thousand executions. It is important to notice that in some machines, even the timeout of 500 ms would also lead to some failures. For instance, we have executed the test with the timeout of 250 ms on another machine and in just one execution of a hundred the test has passed. Therefore, testers end up increasing a lot the timeout, which makes the tests take too much time to be executed as several invocations of wait operations may be necessary in the same test. For instance, we have decided to use a timeout of 3 seconds. However, the mean time of the test executions with a timeout of 3 seconds was of 14.65 seconds, while the test executions regarding the test using 500 ms of timeout took on average 4.66 seconds and the executions that did not fail of the test using a timeout of 250 ms took 3.64 seconds.

In order to avoid the false positives or too long test executions, we have implemented the tests using ThreadControl and its primitives, such as the `waitUntilStateIsReached()`, which demands a previous `prepare(expectedState)` invocation to inform about the expected state and a following `proceed()` operation to release any blocked threads. This version of the tests did not present false positives anymore and was a lot faster than the previous ones as easily observed. Considering the extended version of the `testAddPurchase()` without invocations to `Thread.sleep(timeout)`, the mean time of test executions was 3.15 seconds, which is 78.5% smaller than the version with `sleep` using a timeout of 3 seconds and even 13.43% smaller than the version with sleep using a timeout of 250 ms (which may lead to many false positives in some machines).

One difficulty encountered in the preparation of the tests with ThreadControl was in correctly identifying the expected state (such as discovering that we should verify that the test should wait until when one instance of a given `Runnable` has finished). Although this is challenging, we believe that it is necessary in order to better understand how the system works. When the test team is different from the SUT development team, it is important to get from the developers this information in order to implement the auxiliary methods to be used by tests, such as the `getAddCustomerFinishedState` method presented in the previous section. Specifying the number of times that instances of a certain class have reached a given state is one of the options when using ThreadControl. A simpler option is to request to wait until all instances of a given `Runnable` have finished. However, we could not use ThreadControl in the latter way because the MySimpleCRM system was using thread pools to execute the `Runnables`, which are not yet supported in the current version of the framework while analyzing the state of all instances of a given `Runnable` class.

We have tried to extend ThreadControl in order to provide this functionality. Our main lesson is that dealing with concurrent systems is challenging, and specially

dealing with code that monitors and controls the execution of such systems as ThreadControl does. It is really easy to insert a deadlock in the code if we do not pay attention and really understand what we are doing and how the Java concurrency API works. This is even more challenging when we are using AspectJ, as a single line of this language can affect several lines of the SUT code. After several attempts to extend ThreadControl to support thread pools, we have got a newer version of the framework, but it needs some more tests in order to guarantee that it does not break previous functionalities of the framework. At a first glance and after some executions of its tests, it seems to be working, but a lot more work is still necessary, especially regarding several executions of these tests stimulating different thread scheduling. This makes us conclude that extending ThreadControl, which is not necessary for the test of many systems, is time consuming due to its complexity, specially regarding the testing phase of the framework development. One of the main reasons for that is the use of AOP. However, we believe this is a great use of AOP as it avoids a lot of direct changes on the SUT code.

5. Related Work

There are some previous work about ThreadControl, but mostly focused on the general approach for tests that it supports, the "Thread Control for Tests Approach", on the framework implementation details and on case studies to show that the framework actually avoids false positives [Dantas et al. 2008][Dantas 2010]. In this paper we differ from previous works because we focus on the use of ThreadControl in Information Systems and on the practical experiences of this use and of an attempt to evolve it.

In previous editions of SBSI, some of the closest works are [Veloso et al. 2010] and [Silva-de-Souza et al. 2012]. These papers bring important discussions regarding software testing. [Veloso et al. 2010] discusses how we can compare testing tools, but none of the tools discussed have the same purpose of ThreadControl. [Silva-de-Souza et al. 2012] presents an approach for RESTful Web Service test case generation, focusing on the specification and generation of test cases. However, none of them focus on testing asynchronous systems like ours.

6. Conclusions and Future Work

In this work we describe our experience in using the ThreadControl framework to automatically test an information system with asynchronous operations and how this activity may be performed. Our main observation is that in order to do that we need a good understanding of the behavior of the SUT threads and some auxiliary test methods for creating specifications of desired expected states for system threads. Besides, we have also observed that extending ThreadControl is not so simple and in order to do that, a great knowledge of AspectJ and the Java concurrency API is necessary.

One planned future work is to perform more tests on the evolved version of ThreadControl and explore its use in different distributed systems and in systems where more system-specific waiting states can be demanded by the tests.

References

Boudreau, M. C. e Robey, D. (1999)"Organizational transition to enterprise resource planning systems: theoretical choices for process research" In: International Conference on Information Systems, 1999, Charlotte, North Carolina, USA. Charlotte: ICIS.

Dantas, A., Gaudencio, M., Brasileiro, F. e Cirne, W. (2008) “Obtaining trustworthy test results in multi-threaded systems”. In SBES 2008: Proceedings of the XXII Simpósio Brasileiro de Engenharia de Software.

Dantas, A. (2010) “Aumentando a Confiança nos Resultados de Testes de Sistemas Multi-threaded: Evitando Asserções Antecipadas e Tardias”. Tese (Doutorado em Ciência da Computação) - Universidade Federal de Campina Grande, Campina Grande.

Dustin, E., Rashka, J. and Paul, J. (1999) Automated Software Testing: Introduction, Management and Performance. Addison-Wesley.

Fingar, P., Kumar, H. and Sharma, T. (2000) “Enterprise e-commerce” Tampa, Florida: Meghan-Kiffer Press, p. 359.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1994) Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Garcia, E. e Bazittu, C. (2007) “A Importância do sistema de informação gerencial para tomada de decisões” In: VI Seminário do Centro de Ciências Sociais Aplicadas, 2007, Cascavel. Anais do VI Seminário do CCSA.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. (2001). Getting Started with AspectJ. *Commun. ACM* 44, 10 (2001), 59–65. DOI:http://dx.doi.org/10.1145/383845.383858

Laudon, K. e Laudon, J. (2004)” Sistemas de informação gerenciais: administrando a empresa digital”. Editora Prentice Hall.

Massol, V.(2004).*JUnit in Action*.Ed. Manning.

Oliveira, D. P. R. (1998) “Sistemas de informações gerenciais: estratégicas, táticas, operacionais”. 5ºedição, São Paulo: Atlas.

Sommerville, I. (2006) Software Engineering. Addison-Wesley, 2006.

Souza, R. F. (2004) “Sistemas integrados e comércio eletrônico” Lavras: UFLA/FAEPE.

Stair, R. M., Reynolds, G. W. (2002) Princípios de sistemas de informação. 4°ed. Rio de Janeiro: LTC.

Veloso J. S., Neto P. A. S, Santos, I. S., Ricardo Britto. (2012) “Avaliação de Ferramentas de Apoio ao Teste de Sistemas de Informação”. In: VIII Simpósio Brasileiro de Sistemas de Informação. São Paulo, SP.

Salas, P. P. and Krishnan, P. (2009) Automated Software Testing of Asynchronous Systems. Electronic notes in theoretical computer science, v. 253, n. 2, p. 3-19, 2009.

Silva-de-Souza, T., Correa, A. L., Alencar, A. J. Schmitz, E. A. (2012) “Uma Abordagem Baseada em Especificação para Testes de Web Services RESTful”. In: VIII Simpósio Brasileiro de Sistemas de Informação. São Paulo, SP.

Ventorin, A. J. (2006) “ERP - Enterprise Resource Planning: uma abordagem aos sistemas de gestão integrada” In: Revista Universo Acadêmico, v. 6, p. 7-20.