Test-Driven Development Metodology Proposal for Web Service Choreographies[*]

Felipe M. Besson, Pedro M. B. Leal, Fabio Kon

Department of Computer Science
Institute of Mathematics and Statistics
University of São Paulo (USP)

{besson,pedrombl,fabio.kon}@ime.usp.br

[*]The research leading to these results has received funding from HP Brasil under the Baile Project and from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement number 257178 (project CHOReOS - Large Scale Choreographies for the Future Internet).

1. Introduction

Web service choreographies are a scalable and distributed approach for composing web services in complex business workflows. Differently from orchestrations, which are a centralized approach, the interaction among the choreography services is collaborative with no centralized coordination point. Each participant plays a role specified in a global model that defines the messages exchanged in the collaboration. Thus, choreography is a means to document and monitor Business-2-Business (B2B) interactions from a global perspective [Pel03].

A few standards, such as Web Service Choreography Description Language (WS-CDL), OMG's Business Process Model and Notation version 2 (BPMN2), have been proposed for modeling choreographies. However, up to now, none of them have experienced wide adoption. As a consequence, choreographies are implemented using *ad hoc* development process models. Neither the functional behavior nor scalability of choreographies is assessed properly.

Our goal is to apply Test-Driven Development (TDD) in choreographies to aggregate more discipline in their development to leverage adoption of choreographies. TDD consists in a design technique that guides the development of software through testing [Bec03, Fow11]. Experiments and empirical observations in the industry have shown that TDD applied in traditional software (client-server model) increases code quality and reduces defect density. Another experiment conducted by George and Williams [GW03] with 24 professional developers showed that 87.5% of the programmers believed that TDD facilitates requirements understanding, and 95.8% believed that TDD reduced debugging effort. Later on, a case study [NB006] conducted in Microsoft assessed the impact of TDD in two different teams. In the first one, although the initial development time of the project using TDD increased in 35%, the density of defects decreased 62%. In the second case, the initial development time increases 15%, while the density of defects decreased 76%.

Given this favorable retrospect, this paper presents a TDD methodology proposal for choreographies. The methodology is supported by the Rehearsal framework [BLK11], a tool for applying automated unit, integration, and scalability testing on choreographies. In Section 2, we introduce the Test-Driven Development technique. In Section 3, we present the development activities of a general choreography development model. We present our methodology proposal in Section 4. Finally, we draw our conclusions and the ongoing work to assess the methodology, and the Rehearsal tool in Section 5.

2. Test-Driven Development (TDD)

Test-Driven Development (TDD) consists of a design technique that guides the software development through testing [Bec03, Fow11]. TDD can be summed up in the following iterative steps:

- Write an automated test for the next functionality to be added into the system;
- Run all tests and see the new one fail;
- Write the simplest code possible to make the test pass;
- Run all tests and see them all succeed;
- Refactor the code to improve its quality.

In addition to these steps, according to Astels [Ast03], to apply TDD, developers should follow principles such as: (i) maintaining an exhaustive suite of programmer tests; (ii) only deploying code into the production environment if it has tests associated. Differently from unit tests, which are written to assess a method or class, programmer tests are tests written to define what must be developed. Programmer tests are similar to an executable specification since these tests help developers understand why a particular function is needed, to demonstrate how a function is called, or what are the expected results [Jef].

Having tests associated with the code, gives the developer confidence and courage to make changes and detect immediately (or in a short time) possible introduced problems. Thus, with the absence of tests, it is not possible to assure the correct behavior of the code when it is deployed or integrated into the production environment. Given this importance, in eXtreme Programming (XP) [Bec00], it is often said that a feature does not exist until there is a test suite associated to it.

As a design technique, TDD is not only about software testing but also a learning process. Applying different levels of tests, the development team can clarify the user and customer expectations, and then refine the system requirements [FP09].

3. Web Service Choreographies

A choreography is a collaborative interaction in which each involved node plays a well defined role. A role defines the behavior a node must follow as part of a larger and more complex interaction. When all roles have been set up, each node is aware of when and with whom to communicate, based on pre-defined messages specified by a global model [BBRW09]. Therefore, when the choreography is started (enacted), there is no central entity driving the interaction of the choreography services.

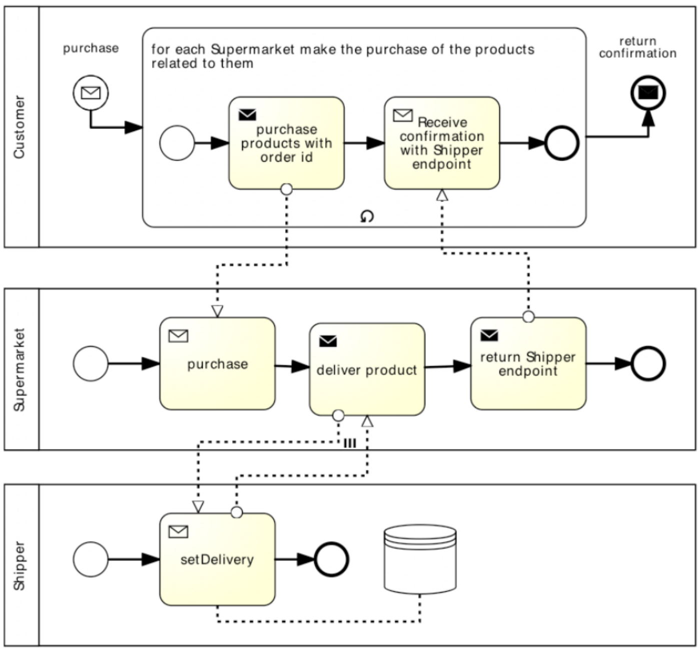

Customer
purchase
for each Supermarket make the purchase of the products related to them
return confirmation
purchase products with order id
Receive confirmation with Shipper endpoint
Supermarket
purchase
deliver product
return Shipper endpoint
Shipper
setDelivery

Figure 1. Choreography specified in BPMN2

In Figure 1, we present an example choreography specified in BPMN2. This choreography is composed of three roles (*Customer*, *Supermarket*, and *Shipper*), which are interacting to execute the *Purchase* operation. Choreographies can be enacted by using distributed orchestrations [Pel03]. This way, at a high level, the global interactions of many participants are specified by the choreography model (as depicted in Figure 1), but each role can be played by composing a service or a set of services into an executable process. In a choreography, a role is well defined by an expected interface and behavior. In our example, the choreography participant playing the *Customer* role must be able to receive the *Purchase* operation, receive from other participant the *Shipper* endpoint, and then return the *Purchase* confirmation to the end-user.

In this approach, the expected role interface can be defined in the Web Service Description Language (WSDL), while the behavior can be specified in the Business Process Execution Language (BPEL). Thus, to play a specific role, a service or a set of services are orchestrated using BPEL, and the WSDL interface exposed by the executable process must match the role interface. This approach is not the only method for implementing the roles of a choreography. As long as the interface matches with the expected role interface, the behavior can be implemented using other approaches such as Service Component Architecture (SCA) or other technologies.

4. Methodology Proposal

The methodology proposed follows the enactment approach presented in the previous section. To apply Test-Driven Development on a choreography, the methodology is supported by the Rehearsal [BLK11]. Rehearsal is a framework developed in Java for automated testing of web service choreographies. During choreography development, Rehearsal will provide features for testing the entire process of composing services. To achieve that, the framework provides the following main features:

- *Abstraction of choreography:* The elements of a choreography such as roles, services, and messages exchanged are represented by Java objects. Through these objects, the developer will interact easily with the choreography elements for writing automated tests in a simple and natural way;
- *Dynamic generation of web service clients:* Given a web service URI, the framework provides mechanisms for invoking the service operations dynamically, avoiding the need to program manually service stubs.
- *Message Interceptor:* the messages exchanged in the choreography can be intercept, and then, validated by the developer;
- *Service mocking:* double services [Mes07] (e.g., a mock) can be created to simulate real services that cannot be accessed or are not available during the tests;
- *Scalability testing support:* Enables assessing the choreography performance and its services as work-load increases.

Based on these features, the TDD methodology proposed for choreographies consists of four phases that are applied iteratively. These phases are depicted in Figure 2. All internal activities belonging to each phase are performed in a testing (development) environment, which can be the developer computer or a cloud infrastructure when the activities demand a large amount of resources.

Depending on the development scenario, the internal activities of each phase are not fully executed. Choreographies can be implemented by partners belonging to different organi-

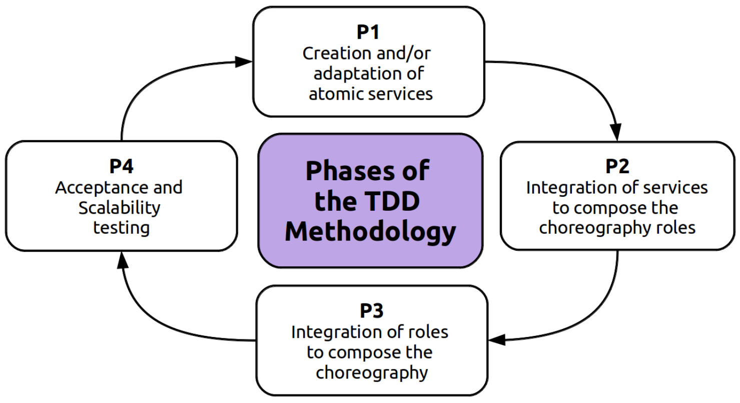

P1
Creation and/or adaptation of atomic services
P2
Integration of services to compose the choreography roles
P3
Integration of roles to compose the choreography
P4
Acceptance and Scalability testing
Phases of the TDD Methodology

Figure 2. Methodology Phases

zations, thus, from a partner point of view, a developer can be involved in different development scenarios. In the first scenario (S1), the developer can be responsible for defining a new role or a set of roles for an existing choreography. In this case new roles must be developed. In the second scenario (S2) the role might already be defined, and the developer is in charge of developing a set of web services to implement this role. Finally, in the third scenario (S3), a partner (e.g., an organization) is beginning to develop a choreography. In this case we can say that the developer implements a choreography, or a part of it, from scratch. In Table 1, we map the phase activities to each scenario, then, we describe in detail the phases in the next sections.

	Scenario 1 (S1)	**Scenarion 2 (S2)**	**Scenario 3 (S3)**
P1	- test-driven approach - unit testing (service interface)	- contract-driven approach - unit testing (service interface)	- test-driven approach - unit testing (service interface)
P2	- service mocking - integration testing (internal message exchange)	- service mocking - integration testing (internal message exchange) - compliance testing	- service mocking - integration testing (internal message exchange)
P3	- role mocking - integration testing (external message exchange)	- role mocking - integration testing (external message exchange)	- integration testing (external message exchange)
P4	- choreography features testing - scalability reached assessment	- choreography features testing - scalability reached assessment	- choreography features testing - scalability reached assessment

Table 1. Activities present in each phase for each development scenario

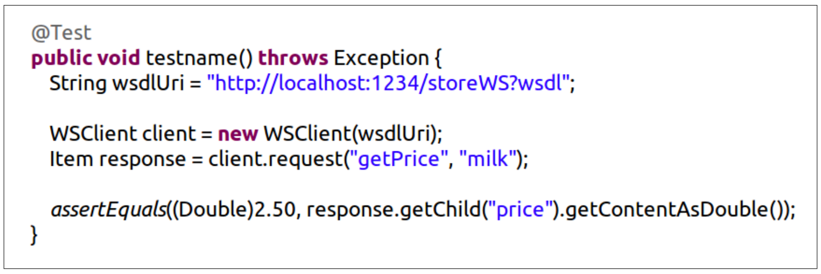
Phase 1: Creation or adaptation of atomic web services

During the implementation of the choreography roles, new web services need to be created or existing ones must be adapted to implement the role requirements. Normally, a contract-driven approach is used for creating or adapting these services. In this case, the service operations have already been defined in a contract, and based on it, the service is coded. In the case of scenarios S1 and S3, instead of using a contract-driven approach, the developer may apply a test-driven approach using the Rehearsal framework.

To invoke the services, developers can use tools, such as Apache Axis[1] and JAX-WS[2].

[1]Apache Axis: `http://axis.apache.org/axis`
[2]JAX-WS: `http://jax-ws.java.net`

With these tools, it is possible to create stub objects (also called clients) from a valid WSDL specification, and then write tests that invoke the service through the stubs. One drawback behind this approach is that without the contract (e.g., WSDL specification), stubs cannot be created and consequently, tests cannot be written, which prevents the test-driven approach. Besides, if the WSDL specification of the requested service changes, existing clients need to be regenerated. To solve this, Rehearsal provides a feature for the dynamic generation of web service clients. With this feature, the developer can interact with a service without creating stub objects. Given a web service interface (in WSDL), its operations can be requested dynamically as depicted in Figure 3.



```
@Test
public void testname() throws Exception {
  String wsdlUri = "http://localhost:1234/storeWS?wsdl";

  WSClient client = new WSClient(wsdlUri);
  Item response = client.request("getPrice", "milk");

  assertEquals((Double)2.50, response.getChild("price").getContentAsDouble());
}
```

Figure 3. Example of dynamic request to a Soap service using the WSClient

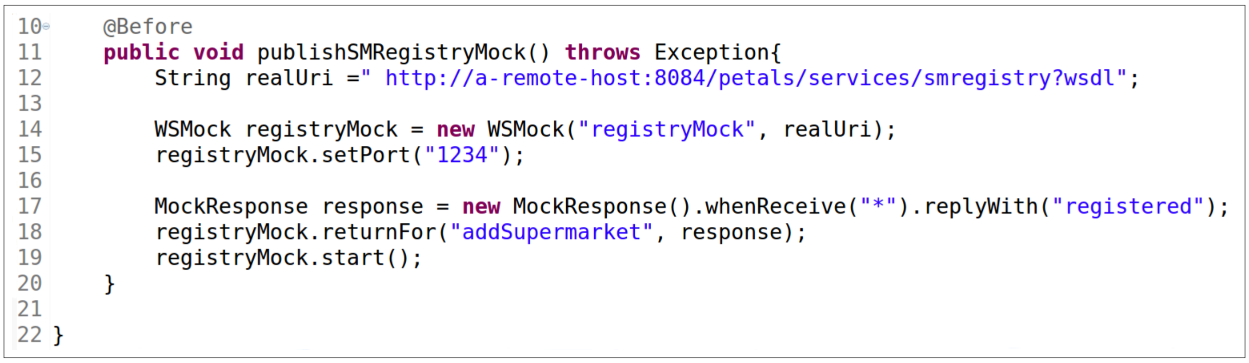
In Figure 3, the service under test does not exist yet. But, using Rehearsal, one can apply a test-driven approach for implementing it. Thus, in the test, the developer can specify the service endpoint (WSDL URI), the operation name (*getPrice*), and signature (receive a *String* and return a *Double* object). After writing the tests, the developer must code the service to make the tests pass. One can notice that even if the developer is involved in a scenario similar to S2, in which the roles are already defined, Rehearsal can still be used, in this case, with *test-after* development approach.

Phase 2: Integration of services to compose the choreography roles

After the services are created (or adapted) and tested properly, they are integrated to compose a choreography role. As explained in Section 3, at this point, a role consists of an executable process defined by a service or a set of services. When a set of services is needed for the composition, third-party services may not be available at development-time. To solve this integration problem, Rehearsal provides a service mocking feature where real services (e.g., third-party ones) can be simulated.

In the example in Figure 4, we are mocking a real registry service, which is published in the URI specified in line 12. For playing the *Supermarket* role (see Figure 1), the service must register its endpoint into a *Registry* service. If this service is only available in the production environment, it can mocked as presented in Figure 4. Thus, we can simulate the service registration for validating the role implementation in the testing environment. Thus, in lines 14–15, we instantiate a *WSMock* object; in lines 17–18, we define the response this object must provide for any parameter received. Finally, in line 19, we start the mock, and its WSDL interface is published on the URI: *http://localhost:1234/registryMock?wsdl*.

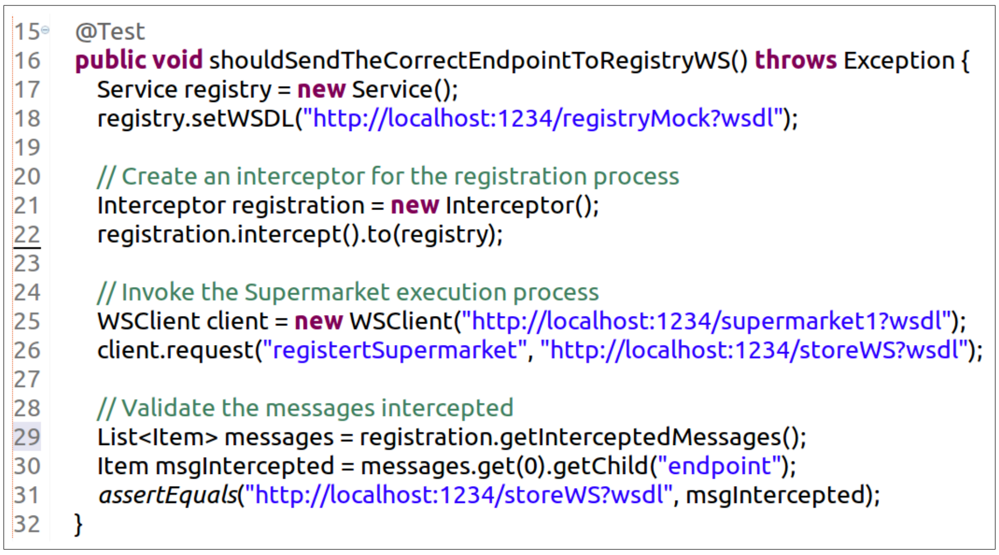
After all real dependencies have been mocked, when it is needed, the services can be integrated to compose a choreography role. To assess the messages exchanged among the



```java
    @Before
    public void publishSMRegistryMock() throws Exception{
        String realUri =" http://a-remote-host:8084/petals/services/smregistry?wsdl";

        WSMock registryMock = new WSMock("registryMock", realUri);
        registryMock.setPort("1234");

        MockResponse response = new MockResponse().whenReceive("*").replyWith("registered");
        registryMock.returnFor("addSupermarket", response);
        registryMock.start();
    }

}
```

Figure 4. Example of web service mocked with WSMock

services within the executable process, Rehearsal provides a message interceptor feature. Using this feature, tests to validate this message exchange can be written before the developer performs the real integration.

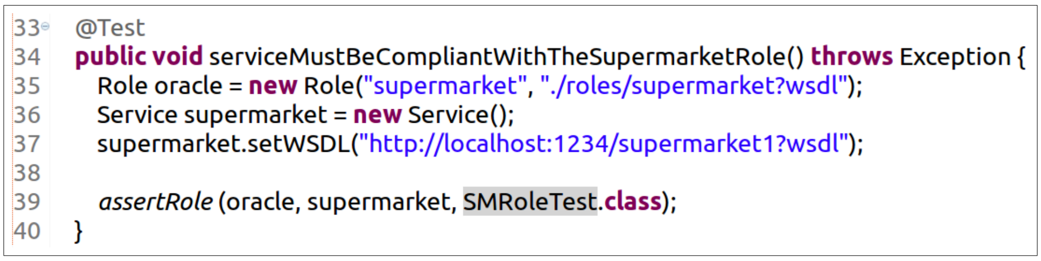


```java
15  @Test
16  public void shouldSendTheCorrectEndpointToRegistryWS() throws Exception {
17    Service registry = new Service();
18    registry.setWSDL("http://localhost:1234/registryMock?wsdl");
19
20    // Create an interceptor for the registration process
21    Interceptor registration = new Interceptor();
22    registration.intercept().to(registry);
23
24    // Invoke the Supermarket execution process
25    WSClient client = new WSClient("http://localhost:1234/supermarket1?wsdl");
26    client.request("registertSupermarket", "http://localhost:1234/storeWS?wsdl");
27
28    // Validate the messages intercepted
29    List<Item> messages = registration.getInterceptedMessages();
30    Item msgIntercepted = messages.get(0).getChild("endpoint");
31    assertEquals("http://localhost:1234/storeWS?wsdl", msgIntercepted);
32  }
```

Figure 5. Example of message intercepted using the Message Interceptor

Figure 5 illustrates a test case that validates the integration of the service assessed on Figure 3, with the *Registry* mock service (see Figure 4) by analyzing the messages exchanged during their interaction. First (lines 17–18), a *Service* object is created to represent the mock created previously. Then, on the lines 21–22, we define a message interceptor to intercept all messages sent to the mock service. To trigger the messages exchanged within the executable process, or in other words, the role implementation, we invoke the process, which is exposed as a service in lines 25–26. Finally, we retrieve and validate the messages intercepted in lines 29–31.

In the case of scenario S2, where the role contract is already defined in the choreography model, the developer can use the contract as an oracle and then validate his/her implementation. The idea is that the role implemented must have the same interface and behavior of the oracle. Rehearsal provides a feature for applying compliance tests which aim at verifying if a service is playing the role properly based on the interface of this oracle contract. As depicted in Figure 6, an oracle specification (line 35) is defined and the service implemented is assessed based on this specification. This assessment is performed by the *assertRole* assertion, which verifies if the service interface matches with the oracle interface. Then, the test cases defined on *SMRoleTest* class are applied on the service: if all tests succeed, the service is in compliance with the role. This class consists of JUnit test cases that interacts with an endpoint which is retrieved from the *assertRole* parameters at runtime. In the example, the endpoint is retrieved from the *Supermarket* object. In our future works, we aim at generating this test suite automatically from the oracle object.



```
33  @Test
34  public void serviceMustBeCompliantWithTheSupermarketRole() throws Exception {
35    Role oracle = new Role("supermarket", "./roles/supermarket?wsdl");
36    Service supermarket = new Service();
37    supermarket.setWSDL("http://localhost:1234/supermarket1?wsdl");
38
39    assertRole (oracle, supermarket, SMRoleTest.class);
40  }
```

Figure 6. Example of compliance tests

Compliance tests assure that the implemented service plays the choreography role properly. Thus, these tests give confidence and courage to the developer: (i) integrates the new role into the production choreography; (ii) refactors the code implemented; (iii) modifies the software in case of requirement change.

Phase 3: Integration of roles to compose the choreography

After implementing an executable process by following the steps of the third phase, the developer can assess the integration of the developed service with the rest of the choreography. Regarding the development scenarios (S1 and S2), the services playing the other choreography roles may not be available at development-time. However, since the service contracts (WSDL interfaces) are defined in the choreography models, through these contracts, the unavailable service can be mocked following the process presented in the previous phase (see Figure 4).

In the case of being involved in the scenario S3, the other roles do not exist at the beginning. Thus, the developer can implement a set of roles following the steps of the third phase by integrating the services iteratively. During this process, since the services are under the developer control, the real services can be used in the integration assessment, otherwise, mocks can also be used at this point.

After all dependency problems have been overcome, the integration among the services is performed, and the external messages exchanged are intercepted and then validated. This process is similar to the process described in the last phase (see Figure 5). However, this time, the messages that must be intercepted and validated are the ones specified in the choreography global model, and not the internal ones. In the example depicted in Figure 1, the messages exchanged among the roles *Customer*, *Supermarket* and *Shipper* must be validated.

Phase 4: Acceptance and scalability testing

To complete the overall process, the choreography must be assessed taking into account properties that affect directly the end-user. Thus, acceptance and scalability assessments are applied.

The first one focus on executing the choreography features to validate its functional behavior from the end-user point of view. The second assessment aims at investigating the choreography performance by analyzing how scalable it can be.

Acceptance testing

Differently from others testing strategies, acceptance tests verify the behavior of the entire system or complete functionality. From the point of view of an end-user, the choreography is available as an atomic service. Thus, the acceptance test validates the choreography as a unit service, testing a complete functionality. In such context, this type of test is similar to the approaches of unit tests and WSClient (see Figure 3) can also be used for testing the choreography.

Scalability testing

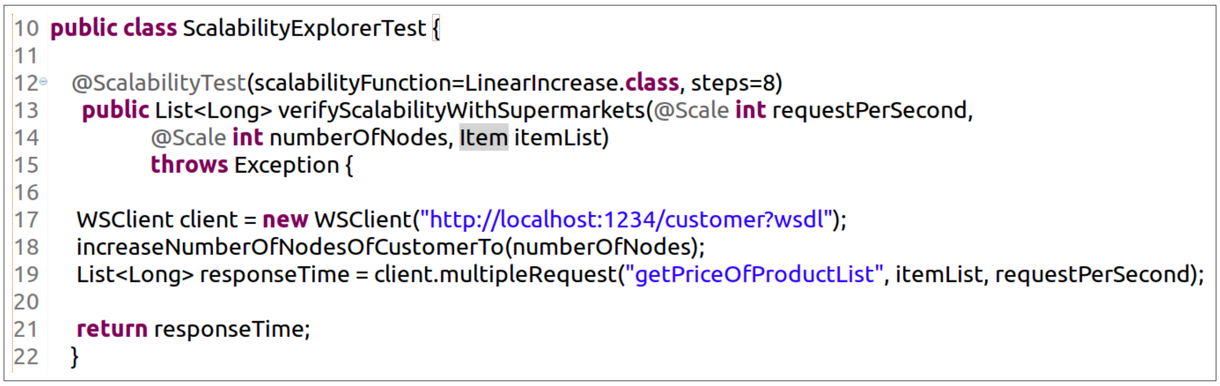
An application is scalable if it achieves the same performance by increasing the architecture capability with the same proportion of the problem size increase. Rehearsal provides the Scalability Explorer component that assists the developer to verify the choreography scalability for future improvements. The developer must apply the following steps to assess a system scalability:

1. Choose the variables that define the problem complexity (e.g., number of requests per second), the performance (e.g., average response time), and the software architecture (e.g., number of nodes of a role);
2. Define the functions of the complexity size of the problem, performance metric, and the architecture capability;
3. Choose initial values for these variables;
4. Execute the application with these initial values for obtaining the initial value of the performance metric;
5. Execute multiple times with the same process and collect the performance metric for each execution;
6. Analyze the performance metric.

The steps 4 and 5 are automated by the Scalability Explorer. It also assists the developer in the Step 6 by illustrating the performance metrics obtained. In the steps 1 to 3, the developer specifies a method using the Java Annotation *@ScalabilityTest* with the parameters: *steps* and *ScalabilityFunction*. The first parameter defines how many times the method will be executed. The *ScalabilityFunction* parameter defines how the method parameters with the *@Scale* annotation will increase at each execution.

Figure 7 illustrates an example of scalability test for the choreography. In the line 12, we specify the *@ScalabilityTest* annotation with the number of steps that the framework will execute increasing the parameters with *@Scale* annotation (*requestPerSecond* and *numberNodes*) linearly for each execution.

The *requestPerSecond* parameter represents the complexity of the problem: if it increases, the complexity size of the problem increases. The *numberNodes* parameter improves



```
10 public class ScalabilityExplorerTest {
11
12   @ScalabilityTest(scalabilityFunction=LinearIncrease.class, steps=8)
13    public List<Long> verifyScalabilityWithSupermarkets(@Scale int requestPerSecond,
14            @Scale int numberOfNodes, Item itemList)
15            throws Exception {
16
17     WSClient client = new WSClient("http://localhost:1234/customer?wsdl");
18     increaseNumberOfNodesOfCustomerTo(numberOfNodes);
19     List<Long> responseTime = client.multipleRequest("getPriceOfProductList", itemList, requestPerSecond);
20
21     return responseTime;
22   }
```

Figure 7. Example of scalability tests

the architecture of the *Customer* role, specified at line 18. If we increase the number of nodes, we improve the architecture.

Since the framework increases both with the same proportion, the performance metric is expected to be constant. The response time, which is the performance metric, for each request is returned as a list (line 19). The Scalability Explorer component collects these lists and calculates the mean and standard deviation for each execution. Finally, the developer can analyze a performance metric graph plotted by the component to verify if it stays constant.

After applying this assessment, the developer can verify how much the choreography is scalable. If it is not scalable enough, the developer can refactor the choreography architecture and analyze again with the same scalability test to verify if the desired scalability was achieved.

5. Conclusions and Ongoing work

Rehearsal framework is still under development. We are working on the message interceptor and scalability testing features. To validate the framework features, we intend to apply quantitative assessments to evaluated aspects, such as time overhead and test case verbosity. In particular, to the scalability testing features, currently, we have a software prototype that consists of the input for defining the final scalability testing API of Rehearsal. To validate our prototype, we aim at assessing the scalability of a test bed choreography we developed.

Regarding the methodology proposed, we intend to apply an experiment involving Computer Science students developing a web service choreography. The goal is to evaluate the user acceptance and the framework effectiveness as well the difficulties faced by the students in the process. To achieve that, a group of users would receive initial and basic training on our testing framework. Possibly, a preparatory short course about choreographies and web services would also be performed. After this training stage, the students must develop simple choreography requirements, possibly a role, using our TDD methodology. At the end, we would analyze the test cases produced as well questionnaires answered by the participants.

References

[Ast03] D. Astels. *Test-Driven Development: A Practical Guide*. Prentice Hall PTR, July 2003.

[BBRW09] Adam Barker, Paolo Besana, David Robertson, and Jon B. Weissman. The benefits of service choreography for data-intensive computing. In *Proceedings of the 7th international workshop on Challenges of large applications in distributed environments*, CLADE '09, pages 1–10, New York, NY, USA, 2009. ACM.

[Bec00] Kent Beck. *Extreme programming explained: embrace change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[Bec03] Kent Beck. *Test-driven development: by example*. Addison-Wesley, Boston, 2003.

[BLK11] Felipe Besson, Pedro Leal, and Fabio Kon. Baile project: Verification & validation(v&v) of choreographies. Available on: http://ccsl.ime.usp.br/baile/VandV, 2011.

[Fow11] Martin Fowler. Test-Driven Development. Available on: http://www.martinfowler.com/bliki/TestDrivenDevelopment.html, 2011.

[FP09] Steve Freeman and Nat Pryce. *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional, 1 edition, 2009.

[GW03] Boby George and Laurie Williams. An initial investigation of test driven development in industry. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 1135–1139, New York, NY, USA, 2003. ACM.

[Jef] Ron Jeffries. What is extreme programming?

[Mes07] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, May 2007.

[NB006] *Evaluating the efficacy of test-driven development: Industrial case studies*, volume 2006, 2006.

[Pel03] Chris Peltz. Web Services Orchestration and Choreography. *Computer*, 36:46–52, October 2003.