Performing Unit Testing Based on Testing as a Service (TaaS) Approach

Lian Yu[1], Shuang Su[1], Jing Zhao[1], Wenbo Zhao[1], Shan Luo[1], Qing Fang[1], Frank Tung[1]

Alice Ying Liu[2], Jun Zhu[2], Hui Su[2]

(1. School of Software and Electronics, Peking University, Beijing 102600, China, lianyu@ss.pku.edu.cn, sushuang0322@gmail.com;
2. China Research Center, IBM, Beijing, 100094, PRC)

Abstract: Unit testing is a fundamental activity of software quality assurance. As it needs extra knowledge and effort than just programming skill, it is either not well carried out or completely ignored. The paper proposes a testing model – Testing as a Service (TaaS), which provides unit testing services to partake the burden of programmers. Bringing TaaS into effect faces a lot of challenges. We put forward to use Ontology technique to model testing knowledge and use SWRL (Semantic Web Rules Language) to describe matching rules between tenant test requests and TaaS capability. Test Service Bus (TSB) is built to recognize the semantics of test requests, map to corresponding test capability based on automatic inference, choreograph testing services and invoke proper testing services. This paper follows a systematic process to identify, specify and realize testing services.

Key words: Unit Test, SOA, Testing as a Service (TaaS), Test Case Generation, Test Environment

CLC number： TP391 **Document code: A** **Article ID :** 123

1. Introduction

Unit testing is a fundamental activity of software quality assurance. It requires extra knowledge and effort than just programming skill. To generate test cases for unit testing, a programmer has to select proper test criteria and codify test criteria into the test cases to verify the correctness of the units of interest.

If not planned carefully, a careless unit test case may execute as an integration test case that involves many software components, and thus preclude the achievement of most if not all of the goals established for unit testing. The objective in unit testing is to isolate a unit and validate its correctness. However, if the unit is dependent on the others; the programmer must prepare the mock objects and test harnesses to assist testing a module in isolation.

Using an automation framework, the programmer must know the ins and outs of the test framework, write domain-related code of test cases, and run test cases. This brings additional efforts to the programmers.

To free up programmers from unit testing burden, this paper proposes a unit testing model – Testing as a Service (TaaS). Unit programmers do not need to write test cases and execute unit testing, but delegate the horse-work to TaaS and just obtain test results. This virtually implements pair-programming model: allowing the programmer dedicating to code-development, and handing code-testing over to TaaS. Thus it results in a fast software delivery.

TaaS also can benefit a project manager who is utilizing the CFI (Call for Implementation: http://cfi.ss.pku.edu.cn), which is a new model of software development process where requirements analysis and design are carried out in-house while the implementation is solicited to community developers. The code collected from community developers must be examined for quality assuranse. Instead of hiring a short-term tester, the project manager would rather think of using long-term established relations with TaaS to give the code once-over of unit testing.

In their forms, TaaS is very similar to Software as a service (SaaS). The latter is a software application delivery model where a software vendor develops a web-native software application and hosts and operates the application for use by its customers over the Internet [1].

SaaS is generally associated with business-oriented software, while TaaS is software development-oriented, and servers for IT participants. Compared with non-IT background users, tenants of testing services are more likely to accept the concept of services. As a third party of testing, TaaS will make them not only shorten time-to-market, but also provide high quality assurance.

The rest of the paper is organized as follows. Section 2 illustrates the architecture of the system based on TaaS approach. Section 3 constructs testing concepts and matching rules using ontology techniques. Section 4 describes the process of identification, specification and realization of testing services. Section 5 presents a prototype of TaaS to show the feasibility. Section 6 surveys the related work. Finally, Section 7 concludes the paper and scratches

the future work.

2. Architecture of Testing as a Service

Fig.1 shows the architecture of testing as a service. The tenants of TaaS can use portals or an integrated development environment (IDE) to access testing services via Internet. In this paper, those portals and IDE are called testing service consumers. Testing service providers supply satisfaction of testing requests from tenants by matching tenants' testing request with TaaS testing capability.

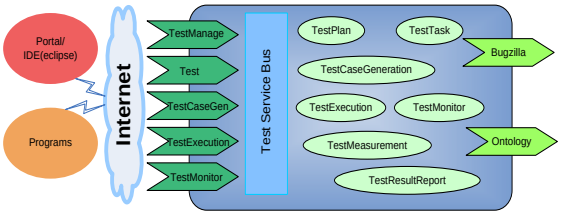

Fig. 1 Architecture of Testing as a Service

Testing Service Consumers of TaaS: There are three ways to allow tenants to access testing services effortlessly: developing Web portals, IDE plug-in, and programmatic access. Tenants interact with Web portals through Internet to create *TestPlan*, manage *TestCase* and *TestSuit*, initiate *TestExecution* and *TestMonitor*; use IDE plug-in to perform Test by button or pop-up windows.

Testing Service Providers of TaaS: TaaS provides two types of services: matching services and testing services.

- TaaS affords the consumers stable testing interfaces, e.g., *TestMangement*, *Test*, and *TestCaseGeneration*. To provide diversity of testing services, Test Service Bus (TSB) takes in charge of recognition of service request semantics and mapping into test tasks, matching test tasks with testing capability of TaaS, and finally invoking corresponding services to satisfy tenant intention. If testing request needs external testing services TSB will route it to proper endpoints.
- TaaS provides inbound testing service and outbound testing services. The former bring direct revenue to a TaaS owner and the latter to another TaaS owner. In this paper, testing services, such as *TestPlan*, *TestTask* and *TestMonitor* are inbound testing services while bug results management is delegated to Bugzilla, and creation of testing ontology and inference are delegated to Protégé [2] and Jess [3] .

3. Matching Testing Services

This section uses ontology techniques to model software testing-related concepts and their relations, and uses SWRL (Semantic Web Rule Language) [4] to model matching rules and other rules.

3.1. Testing Concepts and Relations

A core concept of testing is *TestTask*, which consists of information of *TestActivity*, *TestType*, *TargetUnderTest*, *TestEnvironment*, and *TestSchedule* as shown in Fig.2. *TestTask* represents a tenant request of testing service.

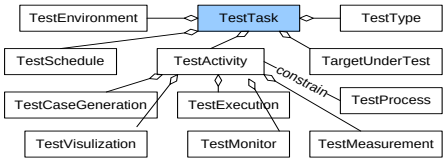

Fig. 2 TestTask Related Concepts and Relations

TestType can be Unit testing, Integration testing and System testing, although this paper focuses on Unit testing. *TestEnvironment* indicates the hardware and software environment to enable a test task, for example, a computer to generate test cases and dependent software package. *TargetUnderTest* contains the target to test on, such as a set of classes, or packages, or a whole system. *TestSchedule* specifies the duration or deadline of time to finish up a test task. *TestActivity* includes *TestCaseGeneration*, *TestExecution*, *TestMeasurement*, *TestMonitor* and *TestVisualization*. *TestProcess* controls the temporal and logical aspects of *TestActivity*.

Another core concept is *TestCapability* as shown in Fig.3, which has a similar structure as TestTask, consisting of *TestType*, *TestActivity*, *TestEnvironment*, *TargetUnderTest* and *QaulityOfService*. However, *TestCapability* replaces *TestSchedule* with *QualityOfService* to indicate the competence to meet test schedule and the degree of tenant satisfaction. If there is no capability that matches a test task requested, the request will be forwarded to Tester, which can be a human being or a team and take proper measure to handle the situation. This paper uses an ontology tool, Protégé to model test concepts.

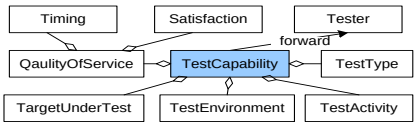

Fig. 3 TestCapability Related Concepts and Relations

3.2. Matching TestTask and TestCapability

A tenant's request represents a set of test tasks and should be fulfilled by TaaS if it has suitable test capabilities, where the relation between *TestTask* and *TestCapability* is "*match*". We propose to use SWRL (Semantic Web Rule Language) to describe the rules of matching based on a set of established basic facts. Each *TestTask* contains several aspects of information which must match corresponding aspects of *TestCapability*.

3.3. Mechanism of Test Service Bus

TSB sits in between tenant requests and testing services. Upon arrival of a tenant request, the role of TSB is to match the test task of request with TaaS test capability, then invoke proper testing services. The mechanism of TaaS is not only for Unit testing, but also adaptive to any other levels of testing. Fig. 4 illustrates the process of TSB.

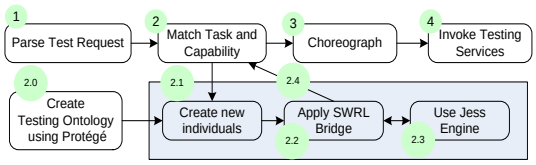

Fig. 4 A Process of Test Service Bus

1. Test request (time, type, schedule, etc.) is parsed.
2. Testing services is selected using SWRL rules reasoning engine.
   2.0. Creating testing ontology using Protégé and exports them into an OWL [5] file.
   2.1. Individuals are created for the SWRL rules, according to the results obtained in step 1.
   2.2. Exporting knowledge bases to Jess [3] rule engine using a SWRL bridge of Protégé
   2.3. Run Jess rule engine to reason matching results of test task and test capability.
   2.4. Exports the reason results further back to the OWL file using the bridge.
3. The invocation sequence of the selected services is choreographed using BPEL.
4. Testing services are invoked according to the choreographed sequence.

4. Constructing Testing Services

This section describes the process and methodology to identify, specify and realize testing services. Focusing on Unit testing, we start from inspecting Unit testing process and identifying each step for possible service candidates, and then specify identified services in a well-defined format, finally implement the testing services. The process is incremental and iterative.

4.1. Identifying Testing Services

Let's examine an example of a unit testing process, which consists of five steps as shown below. We scrutinize each step for possible service candidates in terms of technical feasibility and testing domain necessity.

Create a test plan: Specify a test strategy including unit under test, test type, schedule, environment, techniques to generate test cases, and coverage criteria.

Check UUT (Unit under test) dependency and integrality: Check if the dependents of UUT are available or exist. If not, test stub or mock objects have to be developed to enable the testing process.

Generate test cases: According to the coverage criteria, generate test cases.

Run test cases: Automating test execution is the goal of this step.

Persistence of results: During the execution, test runner captures test failures and stores into database for later analysis or visualization.

Based on authors' knowledge and experiment, it is technically possible to develop from scratch or wrap existing test features as will be demonstrated in Section 5. As a tenant may request any step(s) of testing services at his will, it is necessary to get the services in place. It turns out that activity of each step can become a candidate of testing services in terms of feasibility and necessity.

Different services have different suitable forms exposed to tenants. Creating a test plan needs more involvement of service end users where portal/portlet is more suitable, while the rest of service candidates can be exposed through WSDLs in programmatic. Services can be exposed to external consumers or internal consumers. Not every testing service should be provided inbound, for example, persistence of results can use external services provided by Bugzilla (http://www.bugzilla.org/).

4.2. Specifying Testing Services

This subsection describes the approach to specify testing services in terms of interface, input and output. Take the test steps "check UUT integrality" as examples.

Tab.1 shows the specification of service "check UUT integrality", which takes a jar file and a target class or a set of target classes as inputs. The target classes are the classes which may depend on other classes. The service returns a set of classes that the target class depends on but not in the jar file.

Tab. 1 Service Specification 1

Service	Check UUT integrality
Interface	public String check(String jarFileName, String targetClass)
Input	The jar file of class/classes Under Test
Output	Missing classes, null if no missing

4.3. Implementing Testing Services

Implementing services for Unit testing has three approaches in this research as shown below:

1) Develop from scratch: If there is no required test functionality available or the wrapping is outweighed developing, we choose developing the test services. For example, we developed test case generation and mock objects services from ground-up.
2) Wrap existing test functionality: Existing test functionality is wrapped as testing services. For example, Junit testRunner is wrapped as "Run test cases" service.
3) Integrate basic testing services: If a test task of a tenant has to be fulfilled with more than one basic

testing service, we choose to compose the testing services. The integrated services can be external, for example, bug report is using Web services provided by Bugzilla tool.

5. A Prototype of TaaS

This section shows an implementation of TaaS prototype for Unit testing and discusses the challenges that have been faced during the prototype development.

5.1. TaaS System Architecture

Fig.5 shows the architecture of TaaS prototype developed, which consists of four tiers: presentation tier (Portal and Eclipse Plug-in), TSB (Testing Services Bus) tier, BPEL Tier and service logic tier.

Web Services Portal facilitates tenants to access testing services of TaaS from Web browsers, and Eclipse Plug-in seamlessly amalgamates TaaS with an integrated development environment (IDE).

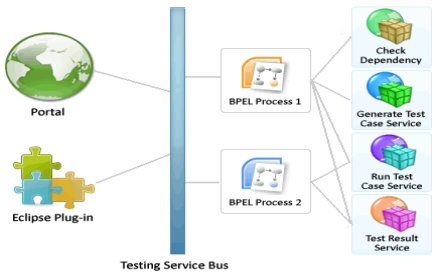

Fig. 5 A TaaS Prototype Design

TSB is similar to ESB (Enterprise Service Bus), and used to integrate applications, coordinate resources, and manipulate information. In this paper, TSB implements the automatic matching process and invokes suitable testing services, which can be one service or a set of services composed by BPEL.

5.2. TaaS Service Design

The following selects to describe three services in the prototype.

Check Dependency Service: This service receives the jar package as data handler, saves the jar file in a temp directory, and finally calls the function of AutoJar [6] to check if the jar file has dependency errors. That is if there is any dependant missed, it will report errors.

Generate Test Case Service: This service can generate test case from an XML file. The most critical step is to parse the XML file with XSLT style sheet defined inside this tool. With all the templates defined inside style sheet, it successfully generates JUnit 4 test cases source code according to the input XML file.

Run Test Cases and View Test Result Service: This service can run test case from xml String, create chart files, and return the file path of the chart files. The process is: 1) Input the test case string (source code); 2) Compile the test cases; 3) Run the test cases using JUnit; 4) Get the test results and record in Database; 5) Generate chart files using Google Chart API and JFreeChart.

5.3. TaaS Services Implementation

The following describes implementations of testing service composition using BPEL, an Eclipse plug-in and a Web services portal using Pluto.

BPEL Implementation: We use Active BEPL Designer 3.1 [7] to design a BEPL process for testing services, where there are two branches. The first one is to generate a test case source code based on an xml file received, and the second one is from an xml file to generate test case, then execute test and finally send the result to the *Test Case Result* service to display a picture on a screen. This workflow is based on the WSDL files with identifying services.

Eclipse Plug-in Implementation: Eclipse is one of the most popular Java IDE. So it is a best choice to develop a plug-in for the Eclipse platform in our TaaS experiment. We add one item to the context menu of .java file in Eclipse and two sub items under it (Fig.6).

When "Run Test" item is clicked, the plug-in will send the test case code to the *Run Test* service and get the result; when "Generate Test Cases" item is clicked on an XML file, the plug-in will send the XML file to the *Generate Test Case* service and get the generated test cases returned as string, upon which the plug-in will generate a .java file of test case code.

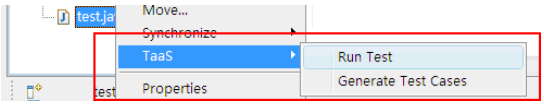

Fig. 6 Eclipse Plug-in Design

Portal Implementation: Apache Pluto is used as the Portal Server. Pluto (http://portals.apache.org/pluto/) is the Reference Implementation of the Java Portlet Specification. The current version of this specification is JSR 168 [8] .

Portal techniques allow us to create many portlets that implement different web services and organize different blocks on the Web page to display them with different information. The prototype contains many portlets which implement the Web Service clients handling the inputs, and display the steps and result of TaaS.

5.4. A Usage Scenario of TaaS Portal

Step 1 – Login: Go to the TaaS Project Web site and login as an authentic account.

Step 2 – Browse Welcome page: There are some instruction and ongoing projects' test results(chart image).

Step 3 – Generate a test case XML file: The first step is to generate a test case file in XML format, which contains test information. Upload the XML file to the TaaS Project web site; and click the "Generate Test Case" button.

Step 4 – Generate test case Java file: This step returns test cases as a Java file, i.e., JUnit 4 test cases and mock objects. Modify the codes if needed, or just click the "Run Test Case Directly" button to run this test case.

Step 5 – View Result Chart: The pass count, failure count, and errors are listed. The Google Chart result shows in the left box and the JFreeChart result shows in the right box as shown in Fig.7. The latest five times of results are shown, where the current result got more passed test cases and less failure compared with the previous one.

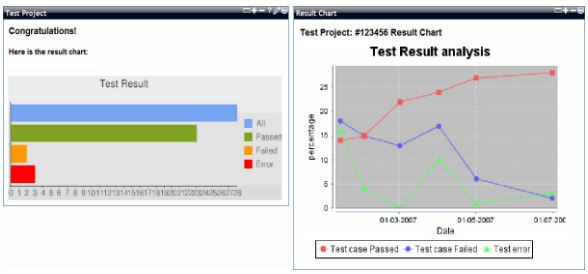

Fig. 7: Testing Results Updated

5.5. Challenge with Implementing TaaS

The implementation of TaaS faces a lot of challenges:

1) It's not easy to create a testing environment on the service provider side, because unit under test usually depends on many other files, databases, and even operating system.
2) Test cases generation needs human being to specify inputs and expected outputs.
3) Network traffic can be high because many files need be transferred. Thus response time may be longer.
4) Pluto and ActiveBPEL do not work together well. We will re-implement these parts.

6. Related Work

SaaS has many good features compared to the traditional model. Services are ordered on-demand which means users have more freedom to choose, and faster to update. Since services are offered through Internet, this makes 7-24 network services possible. Therefore, cost of ownership and marketing costs are greatly reduced [9] .Since SaaS has so many merit, it would be natural to leverage its applications to IT industry itself.

At present, there are some researches on testing Web services [10]; however, little studies focus on applying service-oriented architecture to the domain of testing. In [11], a service-based testing framework is proposed to fill in the gap of traditional testing methods for testing service-based software, and ontology is used to facilitate the solution. In [12], a conformance testing service is created to provide on-line data consistence validation in the implementation of the OpenECG project.

Both of the above works adopt service-based architecture to match some special testing demands and the developed testing services are dedicated, while our aim is to construct an independent service-based testing framework providing general testing services.

7. Conclusion and Future Work

This paper proposed a testing model – Testing as a Service (TaaS). An ontology inference-based approach is proposed to match test tasks of tenants and test capability of TaaS. A testing service bus (TSB) is implemented to intelligently recognize test requests semantics and invoke testing services or combined testing services. Testing services are identified, specified and implemented following a systematic process and applying suitable approaches. A Web services portal and an Eclipse plug-in are developed to facilitate tenants to use TaaS. A TaaS prototype of Unit testing has been developed. To provide a full-fledged TaaS, there are still a lot of works to fulfill: just to name a few, 1) enrich testing services; 2) increase testing performance of TaaS in terms of transferring testing parameters/data through Internet; 3) improve security and trust.

Acknowledgement:

The work presented in this article is partly sponsored by the National High-Tech Research and Development Plan of China (No. 2006AA01Z175) and by IBM Joint Research Program (No. JS200705002). The authors would also thank those who contributed to the design and implementation of TaaS, including Heng Zhang, Jun Wen, Le Zhang, Qing Hao and Guan Zhu Wang.

References:

[1] SaaS: http://en.wikipedia.org/wiki/Software_as_a_Service
[2] Protégé http://protege.stanford.edu/
[3] Jess: http://herzberg.ca.sandia.gov/
[4] SWRL : http://www.w3.org/Submission/SWRL/
[5] OWL: http://www.w3.org/TR/owl-features/
[6] AutoJar: http://autojar.sourceforge.net/
[7] ActiveBPEL: http://www.active-endpoints.com
[8] JSR 168: http://jcp.org/en/jsr/detail?id=168
[9] BRET W. Software as a service: A look at customer benefits. Journal of Digital Asset Management, Volume 1, Number 1, 1 January 2005. 32-39(8).
[10] CANFORA G, PENTA D M, BRERETON P. Testing services and service-centric systems: challenges and opportunities. IEEE Computer Society, IT Professional, Volume 8, Issue 2, March-April. 2006. 10-17.
[11] ZHU H. A Framework for Service-Oriented Testing of Web Services. COMPSAC '06. Computer Software and Applications Conference, 2006. 30th Annual International Volume 2, Sept. 2006. 145 – 150.
[12] CHRONAKI C E, CHIARUGI F, SFAKIANAKIS S, ZYWIETZ CHR. A web service for conformance testing of ECG records to the SPC-ECG standard. Computers in Cardiology, 2005, Sept. 25-28. 961 – 964.