Test-Driven Development of IEEE 1451 Transducer Services and Application

Dušan Marković, Uroš Pešović, Željko Jovanović, and Siniša Ranđić

***Abstract* — IEEE 1451 standard defines the methods of integrating smart transducers into communicating networks. Interface between a user application and a field of transducers, known as Transducer Services API is defined by standrad IEEE 1451.0. This paper presents the use of Test-Driven Design (TDD) in developing methods for accessing transducer services using Transducer Services API and developing web applications which access this services over the network. The characteristics of TDD and its benefits are presented and the way of realization for one method is shown using Java and JUnit framework to run tests.**

***Keywords* — IEEE 1451.0 standard, smart transducers, test-driven developement, Web application.**

I. Introduction

SENSORS have been used in a wide range of applications, such as industrial and home automation, military, healthcare, security, agriculture and environment monitoring. Rapid technological development contributed to the reduction of their dimension and power consumption. Over the time, besides basic measuring functionality, sensors were equipped with other functionalities such as: self-calibration, self-configuration, signal processing. These functionalities provided standalone intelligence and such sensors became known as smart transducers.

In order to measure some event within an area of interest, smart transducers were aggregated into communication networks in order to efficiently collect and present measured data to the user [1]. In early days, networking of such smart transducers was difficult because vendors independently developed communication interfaces which in most cases weren't compatible. Shortly afterwards, a family of IEEE 1451 standards was introduced in order to define standardization of transducers interface and overcome compatibility issues. The IEEE 1451 standard is composed of several sub-standards, among which, the 1451.0 standard defines the common functionalities of transducer network components. It defines the interface used by user applications in order to obtain measured data from deployed sensors in the field. An application can access transducer measured data either by using a HTTP protocol or by Web Services attached transducers.

Traditional software development methods require full specification of the expected system at the beginning of development cycle. In such methods, testing of a developed application was done at the end of development cycle. The development of transducer applications using traditional methods could be difficult, since transducer applications are usually distributed among network entities. Transducers layered structure with high dependency between network entities, poses problems to standard software development methods, especially during testing and debugging phase.

The structure of transducers permits dividing applications into smaller units which could be developed incrementally using agile software development techniques. One of the agile software techniques that could be suitable for developing transducer application is Test-Driven Development (TDD). Such a software development method requires tests execution for each new unit of a code which is developed. Transducers services interface has clearly defined methods that are used to perform the process of measurement and to get the results of measurements. In a specific implementation, only necessary methods could be developed and then subsequently other methods could be added expanding the code in a way that corresponds to incremental development. TDD is capable of replacing undeveloped parts of system with temporal constructors in order to enable tests executions for a unit of code which is currently developed. The aim of this work is to show the capabilities of TDD in the implementation of transducers services and measurement applications. TDD could also be used for creating end user applications for presenting transducers measured data to users and this principle of development was described in [2], [3] and [4].

II. IEEE 1451.0 Standard

The IEEE 1451.0 standard provides a common basis on which transducers interoperability could be established. Using the interface defined by the standard, smart transducers from different vendors could be connected nevertheless on which communication infrastructure they were based. The IEEE 1451.0 standard defines two types of devices which participate in a smart transducer network:

Work presented in this paper was funded by grant TR32043 for the period 2011-2014, by the Ministry of Education and Science of the Republic of Serbia.

Dušan Marković is with the Faculty of Agronomy Čačak, University of Kragujevac, Cara Dušana 34, 32000 Čačak, Serbia (phone: 381-32-303400, e-mail: dusan@tfc.kg.ac.rs).

Uroš Pešović, is with the Technical Faculty Čačak, University of Kragujevac, Svetog Save 65, 32000 Čačak, Serbia (phone: 381-32-302721, e-mail: pesovic@tfc.kg.ac.rs).

Željko Jovanović is with the Technical Faculty Čačak, University of Kragujevac, Svetog Save 65, 32000 Čačak, Serbia (phone: 381-32-302721, e-mail: zeljko.jovanovic@tfc.kg.ac.rs).

Siniša Ranđić is with the Technical Faculty Čačak, University of Kragujevac, Svetog Save 65, 32000 Čačak, Serbia (phone: 381-32-302721, e-mail: rasin@tfc.kg.ac.rs).

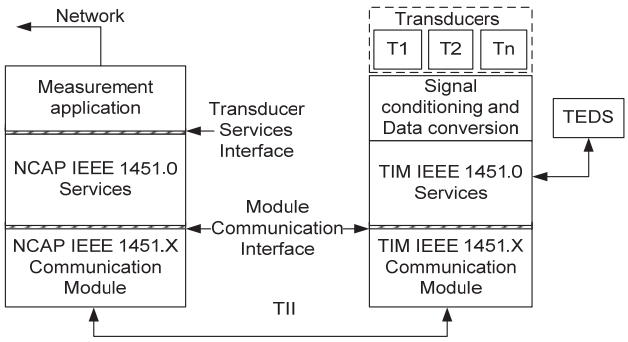

Fig. 1. IEEE 1451 reference model.

Transducer Interface Module (TIM) and Network Capable Application Processor (NCAP) (Fig. 1). TIM is a device that is in direct interaction with its environment via sensors and actuators. Sensors are used to get a measuring signal from physical quantities of interest and actuators are used to affect a specified activity according to a signal they get.

NCAP represents a gateway for a network of TIMs or, in other words, it acts as an intermediary between TIMs and a measurement application. TIM could have one or more *TransducerChannels* depending on a range of logical addresses defined by IEEE 1451 specification. Every *TransducerChannel* could contain converters for signal conditioning and a digital interface for connecting sensors and actuator.

The IEEE 1451.0 standard defines two APIs which are used to connect different transducer network components. Connection between NCAP and TIM was established by Transducer Independent Interface (TII) that defines a communication medium and protocols for data transfer according to standards IEEE 1451.X. Connection between NCAP and TIMs with IEEE 1451.X module is available through a Module Communication Interface. Transducer Services Interface provides access for a measurement application to NCAP services [5]. There are three ways for accessing transducers from the network: 1451.1, 1451.0 HTTP and Smart Transducers Web Services - STWS whose use is recommended in most cases. Service-oriented architecture in the IEEE 1451.0 standard realized through STWS (Fig. 2) is divided into three levels. The first level consists of TIMs which are communicating using the IEEE 1451.X interface with associated NCAP. The second level includes NCAP with attached STWS which enables direct connection to a client application. The third level includes a client application which is structured to fulfill user requirements in representing measured data from field transducers.

STWS could be implemented in three different solutions: an independent Web service (Fig. 2a), STWS as part of NCAP (Fig. 2b), or STWS directly integrated in the transducer which has then a Web services interface (Fig. 2c).

Functionalities offered by web services are described using XML based Web Services Description Language (WSDL). In case of STWS, offered functionalities are defined by the IEEE 1451.0 standard. A user application connects with STWS via SOAP and SOAP/XML

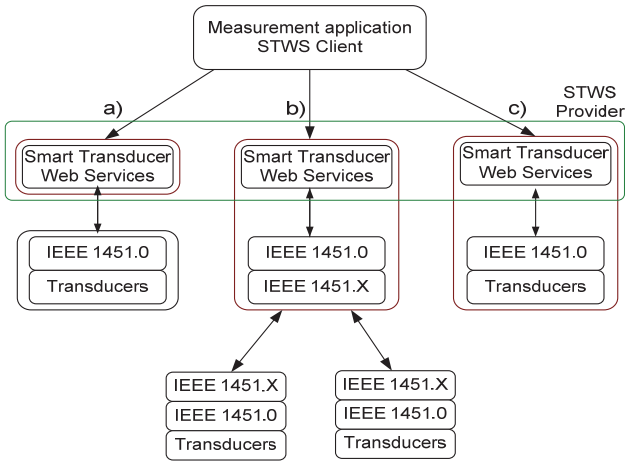

Fig. 2. Service-oriented architecture of smart transducers.

messages that are used to communicate with Web service [1]. After receiving a request from a user application, STWS provider passes it to an appropriate smart NCAP transducer service. NCAP communication module establishes connection to a corresponding TIM module in order to get back results required by the method of transducers service used in that particular request. Finally, STWS provider forwards results back to the user application using SOAP/XML message.

The user application takes values from transducer by accessing its interface. TSI is the standardized interface between the user application and function defined by the IEEE 1451.0 [5]. TSI was subdivided into services according to the following list:

- *TimDiscovery* – provides discovery of all available TIMs with corresponding *TransducerChannels*.
- *TransducerAccess* – represents an interface whose methods are called by application to access *TransducerChannels* sensors and actuators. The most frequently used methods requested by application were methods for read and write on TIM.
- *TransducerManager* – offering more advanced features of smart transducers such as calibration, access right and others. They are placed in this separate service to keep the *TransducerAccess* small.
- *TedsManager* – enable accessing the TEDS and its managing by the application.
- *CommManager* – provides managing of NCAP communication module.
- *AppCallback* – have additional functions such as access to non-blocking I/O operations or measurement streams.

III. Test-Driven Development

Test-Driven Development (TDD) is a discipline of software development where every unit of code is written as a response to a test which must be created first before any production code. In this method, attention is directed to writing formal tests for the smallest increment of functionality. After creating a test, the implementation of working functionality follows with an aim to satisfy the test. These cycles are repeated, so step by step the whole system could be created in this incremental manner.

TDD relies on two basic concepts: test-first and code refactoring. Test-first requires writing a test for small functional units of application before any production code. First, for a small unit of application, a test is written which specifies how this functionality should be integrated in the application and what results could be expected. Then a new test is executed to verify its correctness and, of course, in this phase test execution must not be successful because a working code does not exist yet. Further development goes in the direction of implementing just enough working code to satisfy the test. At the same time, verification of all previously created tests is checked. When the test is successful, then the code is reviewed and design of implemented functionality is improved by refactoring the code.

Refactoring is a process of changing the code for improving its internal structure without influence on the overall functionality from an external perspective. Code refactoring is cleaning from the unnecessary and duplicated constructions which make a code structure too complex [6].

The same steps are repeated and new tests are formulated for other small enough units of code. If these clearly defined steps were followed the program would be incrementally expanded and design also is evolving (Fig. 3). At the beginning of every cycle all tests need to be successful except for a new one which would be used to drive the development of a required production code. At the end of every cycle all tests are run to determine success not only for a new test bur for every test in the test suit which then represents a confirmation that all implemented functionality was obtained as expected [7].

There are some benefits of using TDD that could be pointed out at this moment. TDD helps in understanding a code because a code explanation was given through a test case and a test code provides a direct description instead of more formal documentation. Great efficiency of TDD is a consequence of easier locating of the source of the problem when new defects arise. Quick detection of defects reduces their overall propagation because small changes in one part of the code could be a very possible cause of multiple errors later in other parts of the system. This means that test existence before the implementation of working code facilitates solving the problem of defects propagation that could appear as a consequence of a code error in previous work phases [8].

In traditional techniques, changing a successful working code is usually avoided by code developers because they are afraid of violating code correctness. Using TDD this fear is almost eliminated because code correctness could be checked at the same time by executing all the tests. This means that TDD enables cleaning and restructuring a complex code without fear of accidentally changing code accuracy. TDD technique also guarantees that the internal structure of one part of the system can be changed without the risk of side effects on other part. This characteristic has influence on improving the flexibility of design. The same characteristics improve the self-confidence of code

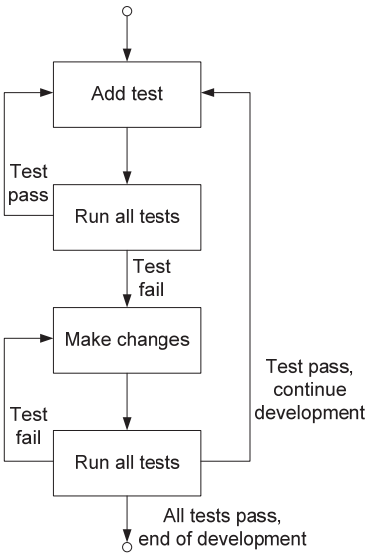

Fig. 3. Block scheme of TDD steps.

developers which means that they are more relaxed and encouraged to introduce more enhancements in the code.

TDD tests unambiguously represent documentation written in a language that programmers can understand. They are so formal that they are executed together with a working code and, because of constant execution, they are never run out of synchronization with application code. Tests represent documentation for low level design which completely describes system interaction and structure, so if it happens to lose a production code, the same one could be created again by examining tests and writing a production code until all the tests are made successful [9].

Using TDD technique, tests were executed very often enabling an error to be detected immediately. The location of an error is well known because it is added together with the last iteration. With TDD, functions that call one another mutually should be apart and tested independently, which is a principle that contributes to better design.

Developers that use TDD are more productive because they are focusing on a small part of the system and their major effort is to satisfy the test of these small units by implementing their production code. Making all decisions is very difficult at the beginning of coding instead of making a decision during code development, which is quite easier and could be enabled by TDD.

Domino effect in software development is well known where a small change in one part could cause unpredictable consequences on other parts of projects. The execution of all existence tests whenever small changes occur is regression testing. Therefore it is very important and represents self-defense against error occurrences. Regression testing could always give a working version of the system at the any moment of adding iterations. According to this, development could be stopped and new requirements could be taken into consideration [6].

Because TDD starts first from a test and carries on through the implementation of a unit's production code, it may happen that a unit requires some other system part that has not yet been developed. The feasibility of this test-first approach is conditioned by having available all other project components that are associated with the tested unit.

To avoid this situation, a temporary construction could be made to play the role of necessary objects and enable applying the TDD approach. Constructions that are used for testing are a *stub* class or *mock* objects. A *stub* class is easy to use and it simply represents the class whose methods need to return data, an expected value for the known input. It has only a small part of overall behavior of the dependency object, needed for unit test. *Mock* objects are simulated objects that take the place of real object and mimic their behavior. With *mock* objects during testing, not only state verification but also behavior verification was performed.

Mock objects have influence on TDD by focusing on interactions with other objects instead of focusing on changes in state. They are used to obtain everything for writing a test that they need from their environment. Using *mock* objects encourages creating a better structure of a test, improving the code, and contributes to clearly defining the interaction between system parts [10].

IV. Development of Transducer Methods with TDD

Methods of TSI defined by the IEEE 1451.0 standard have a precise signature with defined input arguments and returned values. Knowing requirements proposed by the IEEE 1451.0 standard facilitates writing tests for these methods. Smart transducers have a modular structure that could be divided into smaller subparts during implementation which is appropriate for incremental development using TDD.

Java programming tool supports a wide range of additional tools which can be used for automated testing. These tools facilitate creating *mock* (fake) objects and other temporary software constructions used only for testing purposes [11].

JUnit framework is used for applying automated testing for Java programming language. *JUnit* enables direct testing of an isolated area in Java program, hierarchy testing and testing of one or more program units. *JUnit* promotes the idea of writing tests before production code. These tests compare pre-defined expected values with returned results of production code. This framework gives support for automated tests execution while writing a production code until it satisfies particular test [12].

JUnit is a simple and portable tool which complies with the basic Java property of portability. *JUnit* has a graphic interface which can be integrated in Java development environment such as Eclipse. A visual indication of a testing status Testing is represented by a green line for successfully passed tests or by a red line for failed tests (Fig. 4). A test could be created and executed quickly while the results of testing are obtained immediately.

JUnit test (Fig. 4) is composed of two possible test statuses represented with rectangles, and arrows used to show a developer's activity which would cause a change of test status. If all tests were passed successfully, then *JUnit* uses a visual indication of green status line. If any test failed the testing, it will be indicated with a red status line. At the beginning of the development cycle, a test has been written will failed testing since the tested code hasn't

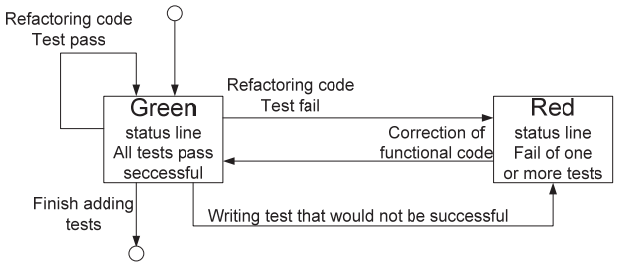

Fig. 4. Visual identification of run tests with *JUnit*.

been implemented yet. This is followed by a red line indication and the code developer needs to write a functional code until the execution of all tests would be successful again. While performing code refactoring of a successfully tested code, any error which will appear in the production code causing the test to fail will be immediately indicated by a red line indication.

A smart transducer service interface is characterized by precisely defined methods that use predefined data types and a known format of input and output parameters. The beginning of the development in our example should be focused on the most needed methods of TSI and then later could be extended according to specific needs.

Usually the most frequently used method is reading measurement data from TIM. As a consequence, an example of this method, which has many earlier forms, was now created according to standard IEEE 1451.0. This method was named *readData* and represents one of the NCAP *TransducerAccess* methods whose implementation was written in Java using also the mentioned *JUnit* framework integrated in Eclipse.

Creating tests and a production code on the NCAP requires some returned values from TIM. Because code development is directed to NCAP side, it would be excessive to simultaneously provide TIM components needed to develop a code using TDD. Independent implementation of one system part is possible without the existence of other associated components that have not been created yet. Instead of real existing dependency parts, TDD could use a temporary construction such as *mock* objects or *stub* class. This enables running all the tests without the need for a completely functional system. It's possible to replace a real system component with an appropriate *stub* class which could be instructed to return predefined values when it's called instead of a real system part.

In our example instead of TIMs there is a *stub* class named *TIMStub* which is used to return predefined values when they are needed during the implementation of TSI methods. In the illustrated example, the method *readData* was presented using TDD approach (Fig. 5).

The first step is writing a test case for *readData* method and setup values that *TIMStub* class should return. The developed test would lead the code developer to the creation of NCAP production code which will satisfy given tests. All tests are executed at once and they gave a clear picture about development progress. Implementation of *readData* code was added incrementally and when TIM values are requested, *TIMStub* class will return predefined

values. Data obtained by *readData* method will be returned to a corresponding test and will be compared to expected values. A test will be finished successfully when there is a match in the result of return *readData* values and expected values in the test and the development of production code covered by its test is completed successfully. The same principle will be applied for all other methods of transducer services.

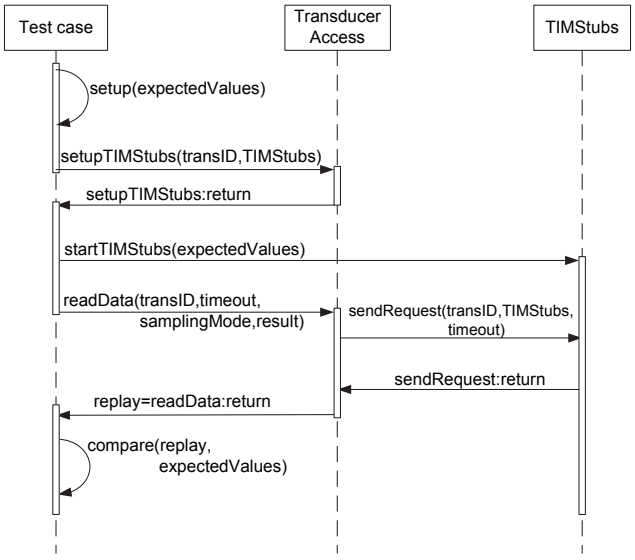

Fig. 5. Example of *readData* method implementing with TDD technique.

Besides transducer services, web applications which access transducer services methods could also be developed using TDD approach. Web services also have an independent structure that could be broken down to compact subparts enabling easy test coverage for these small units of code. For this kind of development, other Java projects components such as Servlet and Java Server Pages (JSP) could be automatically tested using *Cactus* tool. It extends *JUnit* with an additional strategy of testing inside a container that enables running tests for a code that will be deployed on a server side [13].

Cactus is used for testing JSP pages with different tests that could be performed on JSP page. It could be used to verify the result of JSP processing, run tests for JSP tags and testing JSP in the isolation without Java logic which could be replaced usually with mock logic written for tests.

TDD approach could also be used for the development of a user application that could access transducer services via HTTP protocol or by connecting to STWS.

Another component of our system that plays a significant role in obtaining a result from NCAP is STWS. The same principle of TDD could be used to create Web service also using mocking to separate the realization of services methods and Web application. For Web services there is a *SoapUI* testing tool that could be used to make easier implementation base on test driven approach.

SoapUI is an open source testing solution that enables the creation and execution of automated functional, regression or load tests. *SoapUI* is platform independent because it's built on Java with Java Swing GUI. *SoapUI* covers web service invoking, development, mocking and functional testing. Writing functional tests with this tool is accomplished by creating *TestSuites*, *TestCases* and adding assertions by simply selecting objects in a graphical interface [14].

One of the important characteristics of *soapUI* significant for TDD is mocking web services. *MockServices* enables tests execution before the phase in which Web services will be implemented.

In our realization the web application is connected to the Web service and could be developed independently of STWS. For this purpose *MockService* within *soapUI* could be created to simulate STWS that would be implemented in a later phase. Whenever implementing code of the web application requires data from WS to satisfy its test there are pre-defined *MockServices* that would be obtained from these data (Fig. 6).

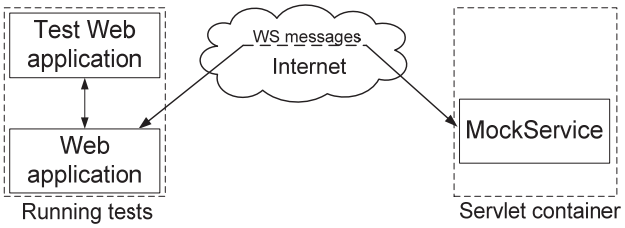

Fig. 6. Using *MockService* for creating web application.

Web services can also be created using *SoapUI* tests, which can drive a production code of web service methods. In the same manner, a test for WS would be written first, after which the development is based on checking values returned from WS methods for a given set of parameters. On the side of *soapUI* a new project would be created followed by *TestSuites* for adding new tests for methods represented as *TestCase* (Fig. 7). For *TestCase* one or more assertions are added to control the returned data in order to drive the development of a production code. This means that pre-defined values posted in *soapUI* would be compared to return values from the appropriate method. When assertion is true, then a test is satisfied which means that invoking WS properly and implementation of WS method is accurate.

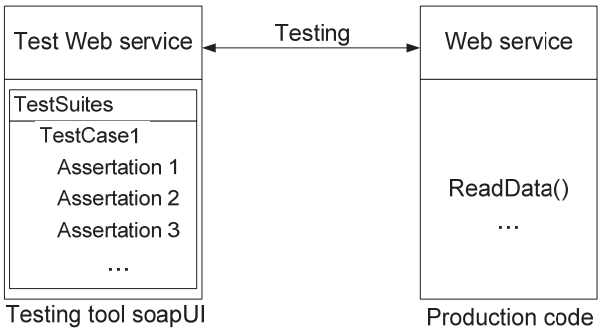

Fig. 7. Testing Web services with *soapUI*.

V. Conclusion

Introduction of the IEEE 1451 standard for smart transducers enables interoperability of transducers from different vendors in a wide range of sensor applications. A

standardized format of transducers service's methods and their incremental development are ideal for software development based on TDD technique. The main focus in this paper was on using TDD technique in developing NCAP services, web application and web services. Wider usage of TDD approach is donated by many software testing tools such as *JUnit*, *Cactus*, *soapUI* which enable much easier test creation and automated test execution. Taking the advantages of these available testing tools in combination with the benefits of TDD technique, all three levels of smart transducers architecture interoperability could be implemented. The benefits of TDD in developing smart transducer applications facilitate developer's work giving satisfaction to programmers by controlling and driving a code with tests.

REFERENCES

[1] E. Y. Song and K. B. Lee, "Service-oriented Sensor Data Interoperability for IEEE 1451 Smart Transducers," *I2MTC 2009-International Instrumentation and Measurement Technology Conference*, May 2009.

[2] P. Hamill, D. Alexander, and S. Shasharina, "Web Service Validation Enabling Test-Driven Development of Service-Oriented Applications," 2009 Congress on Services - I, pp. 467-470, 2009.

[3] D. Tian, J. Wen, Y. Liu, N. Ma, and H. Wei, "A Test-Driven Web application model based on layered approach", ICITIS 2010, pp. 160 - 163, 2010.

[4] J. Burella, G. Rossi, E. Robles Luna and J. Grigera, "Dealing with Navigation and Interaction Requirements Changes in a TDD-Based Web Engineering Approach," vol. 48, Part 2, pp. 220-225, 2010.

[5] *IEEE Standard for a Smart Transducer Interface for Sensors and Actuators – Common Functions, Communication Protocols, and Transducer Electronic Data Sheet (TEDS) Formats*, IEEE Std. 1451.0-2007.

[6] R. Gold, T. Hammell, and T. Snyder, "Test Driven Development: A J2EE Example," 2005.

[7] R. Jeffries, and G. Melnik, "The Art of Fearless Programming," IEEE Software, vol. 24, Issue: 3, 2007.

[8] O. Petter, N. Slyngstad, J. Li, R. Conradi, H. Rønneberg, E. Landre, and H. Wesenberg, "The Impact of Test Driven Development on the Evolution of a Reusable Framework of Components – An Industrial Case Study," *The Third International Conference on Software Engineering Advances*, 2008.

[9] R. C. Martin, "Professionalism and Test-Driven Development," IEEE Software, vol. 24, Issue: 3, 2007.

[10] S. Freeman, N. Pryce, T. Mackinnon, and J. Walnes, "Mock roles, not Objects", OOPSLA '04, 2004.

[11] D. Janzen and H. Saiedian, "Test-Driven Development: Concepts, Taxonomy, and Future Direction," Published by the IEEE Computer Socaity, vol. 38, Issue: 9, 2005.

[12] A. J. S. Mills, "JUnit Testing Utility Tutorial," The University of Birmingham, 2005.

[13] J. Cactus and S. Haines, "Test-driven development for server-side applications", 2009.
Available: http://www.javaworld.com/javaworld/jw-03-2009/jw-03-osjp-cactus.html?page=1

[14] SoapUI, Available: http://www.soapui.org/