Test Points in Self-Testing

Edgars DIEBELIS, Prof. Dr. Janis BICEVSKIS
Datorikas Institūts DIVI, A.Kalniņa str. 2-7, Rīga, Latvia

Abstract. This paper is devoted to the implementation of self-testing, which is one of the smart technologies. Self-testing contains two components: full set of test cases and built-in testing mechanism (self-testing mode). The test cases have been collected since project start and they have been used in integration, acceptance and regression testing. The built-in self-testing mode provides execution of test cases and comparison of test results with saved standard values in different environments. This paper continues the approach described in article An Implementation of Self-Testing, expanding it with the concept of a test point, which allows flexible control of testing actions. Furthermore, the paper describes the first implementation of self-testing using test points.

Keywords. Testing, Smart technologies, Self-testing.

Introduction

The self-testing is one of the features of smart technologies [1]. The concept of smart technologies proposes to equip software with several built-in self-regulating mechanisms, which provide the designed software with self-management features and ability to react adequately to the changes in external environment similarly to living beings. The necessity of this feature is driven by the growing complexity of information systems and the fact that users without profound IT knowledge can hardly use such complex systems. The concept of smart technologies besides a number of significant features also includes external environment testing [2, 3], intelligent version updating [4], integration of the business model in the software [5]. The concept of smart technologies is aiming at similar goals as the concept of autonomous systems developed by IBM in 2001 [6, 7, 8]. Both concepts aim at raising software intellect by adding a set of non-functional advantages - ability to adapt to external situation, self-renewing, self-optimizing and other advantages. However, features and implementation mechanisms of both concepts differ significantly. The autonomous systems are built as universal and independent from properties of a specific system. As a rule, they function outside of a specific system and cooperate on the level of application interface. Hence, we may consider the autonomous systems being more like environmental properties than specific systems. Whereas the features of smart technologies provide a scaffolding, which is filled with functional possibilities of a specific system, thus integrating the implementation modules of smart technologies with the modules of a specific system. Therefore, further development of both concepts is highly valuable.

The first results of practical implementation of smart technologies are available. Intelligent version updating software was developed and is used in practice in a number of Latvian national-scale information systems, the largest of which, FIBU, manages budget planning and performance control in more than 400 government and local

government organisations with more than 2000 users [4]. Firstly, external environment testing [3] is used in FIBU, where the key problem is the management of operating systems and software versions for the large number of territorially distributed users. Secondly, external environment testing is employed by the Bank of Latvia in managing operations of many systems developed independently.. The use of smart technologies has proved to be effective in both cases [9]. The third instance of the use of smart technologies is the integration of a business model and an application [5]. The implementation is based on the concept of Model Driven Architecture (MDA) [10], and it is used in developing and maintaining several event-oriented systems. The use of smart technologies has been proven to be effective according to the results obtained in practical use. This study continues the research of the applicability of smart technologies in software testing.

Self-testing provides the software with a feature to test itself automatically prior to operation; it is similar to how the computer itself tests its readiness for operation when it is turned on. By turning on the computer self-testing is activated: automated tests are run to check that the required components, like hard disc, RAM, processor, video card, sound card etc, are in proper working order. If any of the components is damaged or unavailable, thus causing operation failure, the user receives notification. The purpose of self-testing is analogical to turning on the computer: prior to using the system, it is tested automatically that the system does not contain errors that hinder the use of the system.

The paper is composed as follows: To explain the essence of the self-testing approach, the first section repeats in brief the ideas on the self-testing method and deals with its modes [11, 12]. Section 2 deals with the concept and implementation of test point and Section 3 describes in brief the technical implementation of self-testing.

1. Method of Self-Testing

The main principles of self-testing are:

- Software is delivered together with the test cases used in automated self-testing;
- Regression testing of full critical functionality before every release of a version;
- Testing can be repeated in production, without impact on the production database.

As shown in [11, 12], self-testing contains two components:

- Test cases of system's critical functionality to check functions, which are substantial in using the system;
- Built-in mechanism (software component) for automated software testing (regression testing) that provides automated executing of test cases and comparing the test results with the standard values.

The defining of critical functionality and preparing tests, as a rule, is a part of requirement analysis and testing process. The implementation of self-testing requires at least partial inclusion of testing tools functionality in the designed system. The implementation of self-testing functionality results in complementing the designed system with self-testing functionality calls and a library of self-testing functions (.dll file). Certainly, the implementation of self-testing features requires additional efforts

during the development of the system. However, these efforts are justified by many advantages obtained in development and in long-term maintenance of a high quality system in particular.

The main feature of self-testing is ability to test the software at any time in any environment - development, test and production environments. While developing the mechanism of self-testing the developers may not enter the information into production database; however, they can be used in read-only mode. Hence, it is possible to implement testing in test or production environment without any impact on system use. Of course, it is useful to complement the set of tests with recent system modifications to ensure that testable critical functionality in self-testing is covered.

1.1. Self-Testing Software

The self-testing software is partly integrated in the testable system, which has several operating modes; one of them is self-testing mode when an automated execution of testing (process of testing) is available to the user. After testing, the user gets a testing report that includes the total number of tests executed, tests executed successfully, tests failed and a detailed failure description. The options provided by self-testing software are similar to the functionality of testing support tools.

1.2. Phases of System Testing

In order to ensure development of high quality software, it is recommendable to perform testing in three phases in different environments [11]:

- Development environment - in this environment the system has been developed, errors are corrected and system patches are made;
- Test environment - this environment is used for integration testing, error corrections and improvements.;
- Production environment - this environment is used by the system users. Patches and improvements are set only after obtaining successfully testing results in development and test environments.

Testing phases are described in detail in the article Self-Testing - New Approach to Software Quality Assurance [11].

1.3. Modes of Self-Testing

As shown in [11, 13], the self-testing functionality can be used in the following modes:

- Test storage mode. In this mode, new test cases are defined or existing test cases are edited/deleted. The system logs all necessary information of reading-writing and managing actions by recording them into the test storage file.
- Self-testing mode. In this mode, automated self-testing of the software is done by automatically executing the stored test cases. Test input data are read from the test file.
- Use mode. In this mode, there are no testing activities – the user simply uses the main functionality of the system.
- Demonstration mode. The demonstration mode can be used to demonstrate system's functionality. User can perform system demonstrations, by using stored uses cases in storage files.

Test points are used in the test storage, self-testing and demonstration modes, and they are described in detail in the following chapter.

2. Test Points and Their Implementation

A test point is a command upon which system testing actions are executed. To be more precise, a test point is a programming language command in the software text, prior to execution of which testing action commands are inserted. A test point ensures that particular actions and field values are saved when storing tests and that the software execution outcome is registered when tests are executed repeatedly. By using test points, it is possible to repeat the execution of system events.

As described in the sections above, the self-testing features are introduced in the tested system, namely - written by the test points, which can be introduced in the system in at least two ways:

- By altering the system software's source code. When developing the system, the developer implements in the software code also test points that register system's actions.
- The specialist who defines the business process schemes specifies the test points in the business process. In this case, the business processes and the software must be compatible, and extra resources for moving the testing actions to the software are required. For the time being, the authors do not have knowledge of any instances of application of the approach described above in practice.

When initially developing the self-testing software concept, it was planned to develop only test points that ensure the registration of data storage in the database and data selection from database events. It was important to check whether when executing repeatedly a database command (INSERT, UPDATA, SELECT, procedure or function call etc), the result saved in the database or selected from the database matches the data storing or data selecting performed in the first time.

While evolving the self-testing concept, the idea to use the test point approach to register all system events emerged. Thus, test points register not only data storing in database events or data selection from database events but also other application events (filling in fields in application form, calling application events etc). Such changes ensure that user interface and business logics are tested as well; also, this approach provided a possibility for users to use the system in the demonstration mode. Consequently, with comparatively low investments, the functionality of self-testing was increased considerably.

2.1. Implementation of Test Points

To implement test points in the tested system, it is required to determine how the system works, what is the structure and model of the developed system and what information that characterises the test example should be recorded when registering test examples. It is necessary in order to:

- Be able to develop new or use the existing test points that register the information necessary in the test example files in the tested system;

- Identify where exactly in the software code test points need to be placed to achieve that the critical functionality of the system is covered.

In the self-testing software there are implemented test points that add to the test example scenario the respective test point type object (Figure 1. Adding Self-Testing Test Point Objects to Test Example Scenario).

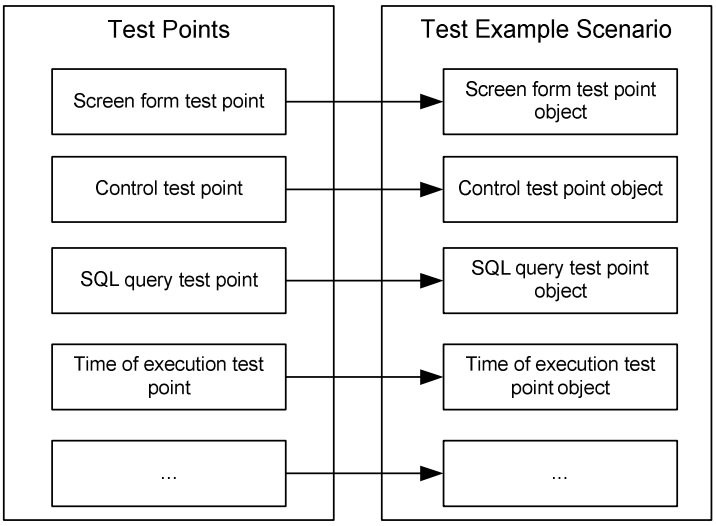

Figure 1. Adding Self-Testing Test Point Objects to Test Example Scenario

Each test point has an additional class that contains all the required information that is received by the test point. Test point objects are developed to make the work related to the test example scenario, developing the scenario and its playback easier. Currently the test point object classes specified in the figure below (Figure 2. UML Class Diagram of Test Point Objects) are implemented in the self-testing software, and they are used to register in the tested system the actions performed by the user or the system.

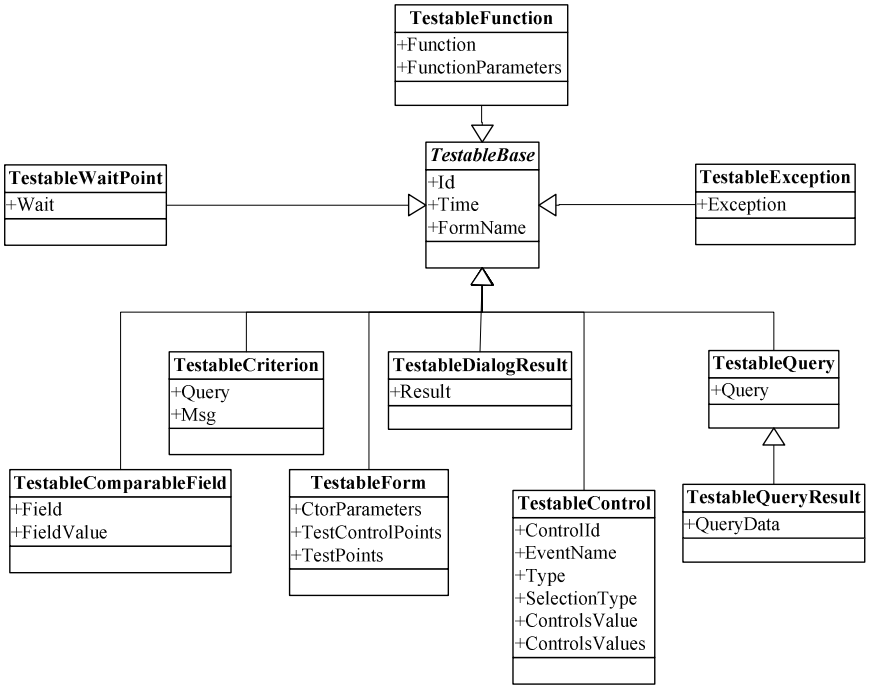

Figure 2. UML Class Diagram of Test Point Objects

The self-testing software employs the following testing actions:

- Opening and closing screen forms (TestableForm). This test point registers in the test example file screen opening and closing actions. The test point type object contains the screen form name;
- Comparable value (TestableComparableField). This test point registers the values of different variables, functions and calculations that are determined in the system code and that the user does not register directly in the system. This test point is necessary to be able to register and compare the values calculated in the system. The test point can be used when the application contains a field whose value is calculated considering the values of other fields, values of which are not saved in the database. Furthermore, the test point can be used in the testing of external interfaces. The test point provides the registration of the values read from the external interface or to be delivered to it in the test example file. In the self-testing mode, operation of the external interface is simulated by reading from or delivering to the external interface the data registered in the test storage mode in the test example file;
- Test execution criterion (TestableCriterion). This test point ensures that test criteria are stored. This test point controls whether it is possible to execute the test (e.g., prior to debiting money from the account, it is checked that the account is not closed and that there is money in it). By using test execution criteria test points, it is possible to specify the criteria for the execution of the stored test. In the system self-testing mode, the test execution criteria points

check whether the conditions specified in the test points are fulfilled. If the criterion is not fulfilled, the test has failed and the user can access a detailed description of test execution, in which the reason for non-execution is specified.

- Form control (TestableControl). This test point is required to register any events performed in the application, e.g. clicking on the button Save or field filling-in event.
- Dialog window (TestableDialogResult). This test point registers in the test example file the results returned by the dialog window. If the user performs actions in the dialogue window, they are registered in the test example file;
- Function (TestableFunction). This test point registers in the test example file the function call, function parameters and the result returned by the function. Executing a test that contains a function test point or a function call, the parameters delivered to the function and the result returned from the function match the information registered during test storage.
- SQL query (TestableQuery). This test point is used to register in the test file the query sent to the database;
- SQL query result (TestableQueryResult). This test point registers specific values that can be selected with an SQL query and that are compared in the test execution mode with the values selected in the test storage and registered in the test file. The SQL query test point can be used after data have been saved to compare the data saved in the database, the data saved when registering the test and the data saved when performing the test repeatedly;
- Time of execution (TestableWaitPoint). This test point ensures the waiting for the execution of time consuming system processes prior to executing further actions;
- System error processing (TestableException). This test point is required to register any system errors that have occurred during software operation or playback. Test points responsible for the creation of objects of this type are added to places where software errors are processed.

Every test point call adds in the software code the respective test point type object to the test example scenario (Figure 3. Test Point Operation Process). Unlike other test points, the screen form control test point in the test example scenario can rewrite the previous screen form control test point. If the action to be registered in the test point takes place in one control, e.g. in the test field value ‘a’ and then value ‘b’ are entered and they together create ‘ab’, then in the test example scenario it is registered that value ‘ab’ has been entered in the test field.

In every test point function that registers in the test example file the information required for the test example there is implemented a check that identifies the mode in which the system operates. If the system operates in the use or demonstration mode, then the test point functions terminate their operation immediately after calling. If the system is used in the test storage or self-testing mode, the test point functions in the test example file register the information that characterises the test example.

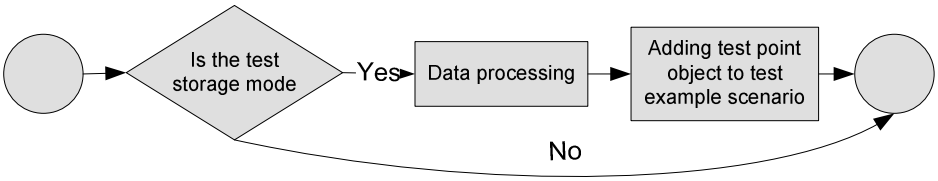

Figure 3. Test Point Operation Process

To ensure conveniently manageable and usable maintenance of test examples, they will be registered and stored in XML files.

2.2. Using Test Points in Modes of Self-Testing

Test points are placed by the developers in the system to achieve that the critical functionality of the system is covered. Test points are used as follows:

- Test storage mode. When the user creates a new test, the specified information in the test file and obtained in testing is registered in the test points implemented in the system. Various types of information can be registered in test points, e.g. value of filled-in fields, clicking a command button, selecting a value from a list etc.;
- Self-testing mode. The software automatically executes the events registered in the test files, replacing the events entered during storage with their selection from the test file. The test points placed in the system during execution of tests create the same test file as in the test storage mode. When the testing is finished, the file created in the test storage mode is compared with the test file created in the self-testing mode. If the contents of the files match, the test has been successful; if they do not match, the testing has failed;
- Demonstration mode. In the demonstration mode, the test files that have been created in the test storage mode and successfully executed in the self-testing mode are used. In the demonstration mode, within a defined time interval or when the user executes commands from the test file step by step, the functionality of the system can be demonstrated both to teach new system users and to demonstrate the system functionality to any potential its buyers.

2.3. Example of Test Point Use

To show how test points are used, a stock purchase transaction process is shown in the next figure (Figure 4. Stock Purchase Transaction Process). The registration of a stock purchase transaction consists of the following main steps:

- Specifying the client;
- Selecting the stock;
- Specifying the number of stocks;
- Saving the transaction.

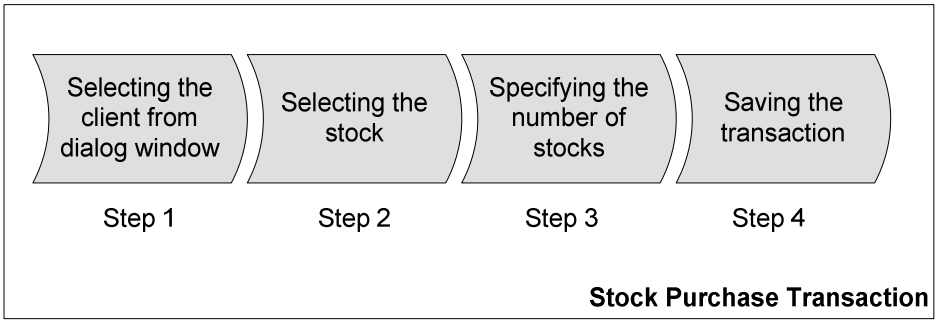

Figure 4. Stock Purchase Transaction Process

To implement self-testing in the stock purchase transaction process, the system would have the following five test points, which various testing actions are written to:

1. Test point *Dialog window* registers the client selected in it in the test storage file (Step 1).
2. Test point *Form control* registers in the test storage file the stock specified for the transaction (Step 2).
3. Test point *Form control* registers in the test storage file the quantity of stocks specified for the transaction (Step 3).
4. Test point *Form control* registers in the test storage file the event of clicking on the button Save (Step 4).
5. Test point *SQL query result* registers in the test storage file the data saved in the database after clicking on the button Save (Step 4).

Classification of test points is outlined in detail below in this Section.

When a stock purchase transaction test case is registered, each of the points in the test storage file registers information that is used to play back the test. When a stock purchase transaction test is plaid back, the self-testing software, step by step, reads from and executes the actions registered in the test file. When the actions specified in the test file are executed, a new test file is created. When all the actions have been executed, the test files are compared; they should match if the tests have been successful. If the files do not match, the user is able to identify in the testing software application the point (command) in the test file that has been executed with errors.

3. Self-Testing Software

The self-testing software is part of the system to be developed. It means that there is no need to install additional testing tools for system testing at the system developers, customers or users. System testing is done by the self-testing software that is partially integrated in the tested system.

Key components of the self-testing software are:

- Test control block. Users use the test control block to perform various basic operations related to test registration and playback;
- Library of test actions. The library contains testing functions that register in the test file the testing actions performed;

- Test file (XML file). This file contains all the required information about the registered test example.

3.1. Test Control Block

The first version of the test control block has been developed. The test control block has been developed with additional functionality and improved with user interface.

The test control block consists of two modules.

- Test control module. The control module is responsible for the control of test examples: it makes it possible to load test examples and delivers tests for execution;
- Test playback module. The test playback module is responsible for test playbacks and for notifying test execution results.

3.1.1. Test Control Module

The control module is a program with a user interface that ensures test management and test execution and the comparing of results (Figure 5. Test Control Module).

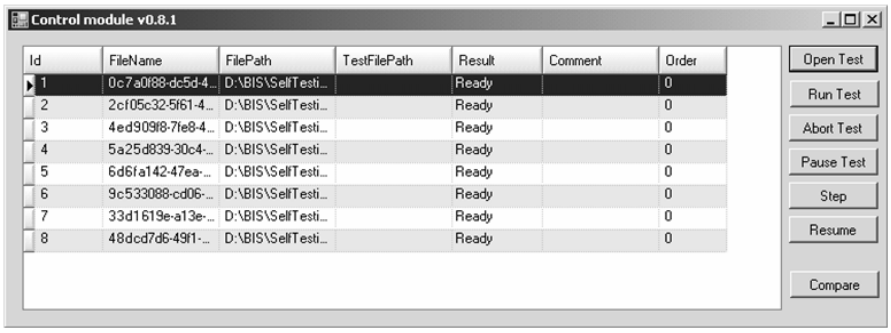

Figure 5. Test Control Module

The control module ensures simultaneous loading and execution of several test examples. Also, the control module determines the succession in which test examples are executed. Simultaneous execution of test examples provides the possibility to simulate simultaneously actions of a number of users in the testing system. The obtained data on the execution of test examples (test execution time) can be used to analyse the system's speed. Successive test example execution can be used if it is required to check several tests as one entire test.

The test control module provides the following functions:

- Delivering the test execution commands initiated in the control module to the test playback module;
- Management of test examples. Loading of test examples and delivering tests for execution;
- Defining the configuration of the self-testing software:

    - Selecting the system operation mode (test storage, self-testing, use, demonstration);
    - Selecting the test mode. The user can specify whether tests should be executed in visible or invisible mode. The visible mode is intended for demonstrations; but if the user wants, they can follow test execution step by step. The invisible mode provides for faster test execution;
    - Feature that ensures the possibility in the test storage mode to register in the test example file all the data returned from the database;
    - Feature that ensures the possibility in the self-testing mode to use the data registered in the test storage mode in the test example file. This functionality makes it possible to repeat the test example execution under the same conditions that were in place when the test was stored. The condition for the use of the feature – when the test example is registered, a feature that registers all the data returned from the database in the test example file must be selected.
- Information on test execution. If the test fails, the control block will provide the user with information on reasons for the failure;
- Deleting tests and test files.

3.1.2. Test Playback Module

The test playback module is responsible for playing black test examples. The module is independent from the test control module. It ensures continuous operation of the control module also in the cases when an error occurs in the test playback module.

The test playback module is a console program, which receives as operation parameters the test file and the program channel name. The program channel is used to ensure interactive playback of the test. Through the program channel, commands from the control module are delivered to the test playback module. The test playback module provides the following key functions:

- Receiving commands from the control module;
- Abort test;
- Pause test. This function can be used to record a new test or to suspend the system during demonstration in order to tell the interested people on a particular system functionality in detail;
- Resume test. This operation is possible if before it the Pause Test or Step operation has been performed;
- Step. This operation stops the automatic execution of the test and ensures that the test can be played back step by step (by test point). This operation can be used in both the demonstration and self-testing modes;
- Create new from existing point. This operation registering a new test example based on an already stored test example. When this operation is executed, a new test example file is created; in its beginning a reference to the related test example file and test points to be executed is recorded.

The result from the test playback module is returned to the control module as a string that contains several elements:

- Test result: the test has been successful or failed, the related test file has not been found or an error of the self-testing software;
- Path reference to the new test file, which was created during test execution.

3.2. Library of Test Actions

The library contains the test action functions described herein. Testing action function calls are implemented in the tested system. Test functions are assigned parameters that characterise the test action. Testing functions, on the basis of the received parameters, make the respective records in the test file.

3.3. Test File

Test file (XML file). Test functions in XML file, using a particular structure, register the values that characterise the test case. The XML file structure and example are described in detail in the article An Implementation of Self-Testing [13].

4. Conclusions

In order to present advantages of self-testing, the self-testing features are integrated in a large and complex financial system. Although efforts are ongoing, the following conclusions can be drawn from the experience:

1. Introduction of a self-testing functionality is more useful in incremental development model, especially gradually developed systems and systems with long-term maintenance and less useful in the linear development model.
2. Self-testing significantly saves time required for repeated testing (regression) of the existing functionality. This is critical for large systems, where minor modifications can cause fatal errors and impact system's usability.
3. Self-testing requires additional efforts to integrate the functionality of self-testing into software, to develop critical functionality tests and testing procedures.
4. The introduction of self-testing functionality would lower maintenance costs and ensure high quality of the system.
5. Self-testing does not replace traditional testing of software; it modifies the testing process by increasing significantly the role of developer in software testing.
6. Test points make test recording and automatic execution much easier. Test points ensure that tests can be recorded in a convenient and easy-to-read manner.
7. Test execution criteria test point determines the possibility to execute the test using the available data set.
8. If test execution criteria test points are used, it is not necessary to maintain the data set which was used to register the test.
9. If test points are used, the user can, independently from the developer, register and then repeatedly execute test cases.
10. Test execution criteria test point provides a possibility to execute tests in random order.
11. The use of self-testing is simple, and system developers should not be afraid of using self-testing.

References

[1] Bičevska, Z., Bičevskis, J.: Smart Technologies in Software Life Cycle. In: Münch, J., Abrahamsson, P. (eds.) Product-Focused Software Process Improvement. 8th International Conference, PROFES 2007, Riga, Latvia, July 2-4, 2007, LNCS, vol. 4589, pp. 262-272. Springer-Verlag, Berlin Heidelberg (2007).

[2] Rauhvargers, K., Bicevskis, J.: Environment Testing Enabled Software - a Step Towards Execution Context Awareness. In: Hele-Mai Haav, Ahto Kalja (eds.) Databases and Information Systems, Selected Papers from the 8th International Baltic Conference, IOS Press vol. 187, pp. 169-179 (2009).

[3] Rauhvargers, K.: On the Implementation of a Meta-data Driven Self Testing Model. In: Hruška, T., Madeyski, L., Ochodek, M. (eds.) Software Engineering Techniques in Progress, Brno, Czech Republic (2008).

[4] Bičevska, Z., Bičevskis, J.: Applying of smart technologies in software development: Automated version updating. In: Scientific Papers University of Latvia, Computer Science and Information Technologies, vol .733, ISSN 1407-2157, pp. 24-37 (2008).

[5] Ceriņa-Bērziņa J.,Bičevskis J., Karnītis Ģ.: Information systems development based on visual Domain Specific Language BiLingva. In: Preprint of the Proceedings of the 4th IFIP TC 2 Central and East Europe Conference on Software Engineering Techniques, CEE-SET 2009, Krakow, Poland, Oktober 12-14, 2009, pp. 128-137.

[6] Ganek, A. G., Corbi, T. A.: The dawning of the autonomic computing era. In: IBM Systems Journal, vol. 42, no. 1, pp. 5-18 (2003).

[7] Sterritt, R., Bustard, D.: Towards an autonomic computing environment. In: 14th International Workshop on Database and Expert Systems Applications (DEXA 2003), 2003. Proceedings, pp. 694 - 698 (2003).

[8] Lightstone, S.: Foundations of Autonomic Computing Development. In: Proceedings of the Fourth IEEE international Workshop on Engineering of Autonomic and Autonomous Systems, pp. 163-171 (2007).

[9] Bicevska, Z.: Applying Smart Technologies: Evaluation of Effectiveness. In: Conference Proceedings of the 2nd International Multi-Conference on Engineering and Technological Innovation (IMETI 2009), Orlando, Florida, USA, July 10-13, 2009.

[10] J. Barzdins, A. Zarins, K. Cerans, M. Grasmanis, A. Kalnins, E. Rencis, L.Lace, R. Liepins, A. Sprogis, A.Zarins.: Domain Specific languages for Business Process Managment: a Case Study Proceedings of DSM'09 Workshop of OOPSLA 2009, Orlando, USA.

[11] Diebelis, E., Takeris, V., Bičevskis, J.: Self-testing - new approach to software quality assurance. In: Proceedings of the 13th East-European Conference on Advances in Databases and Information Systems (ADBIS 2009), pp. 62-77. Riga, Latvia, September 7-10, 2009.

[12] Bičevska, Z., Bičevskis, J.: Applying Self-Testing: Advantages and Limitations. In: Hele-Mai Haav, Ahto Kalja (eds.) Databases and Information Systems, Selected Papers from the 8th International Baltic Conference, IOS Press vol. 187, pp. 192-202 (2009).

[13] Diebelis, E., Bičevskis, J.: An Implementation of Self-Testing. In: Proceedings of the 9th International Baltic Conference on Databases and Information Systems (Baltic DB&IS 2010), pp. 487-502. Riga, Latvia, July 5-7, 2010.