Proceedings of the Ninth International Baltic Conference – Baltic DB&IS 2010

An Implementation of Self-Testing

Edgars Diebelis, Prof. Dr. Janis Bicevskis

Datorikas Institūts DIVI, A.Kalniņa str. 2-7, Rīga, Latvia
Edgars.Diebelis@di.lv, Janis.Bicevskis@di.lv

Abstract. This paper is devoted to the analysis of advantages and implementation mechanisms 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 Self-Testing - New Approach to Software Quality Assurance, expanding it with the concept of a test point, which allows flexible defining of execution points of testing actions. Furthermore, the paper describes the first implementation of self-testing using test points.

Keywords: Testing, Smart technologies, Self-testing.

1 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 looks at the approach for determining the state of database prior to system self-test, and Section 3 deals with the concept of test point, and Section 4 describes in brief the technical implementation of self-testing.

As of writing this paper, the first version of the self-testing software has been developed; it contains a test control block (test execution and test result control) and a self-testing software library, which contains the functions included in the system to be tested. Technology of self-testing could be approbating for different applications. Now the self-testing technology is being approbated for using in securities and currency accounting system applications in banking.

2 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.

2.1 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.

2.2 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 to test error corrections and improvements. In order to replicate situations in the production environment in test environment, at least, for example, once a month production environment should be renewed from a backup in test environment;

- 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].

2.3 Modes of self-testing

As shown in [11], 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. To provide the self-testing mode in the production environment, an additional database, in which test cases are registered and executed, is used. In the case of production environment, during test storage mode the real database is accessible in the read-only mode. Neither development, nor testing environment requires an additional database, since one database is used for both storing and playing back tests. The results of system operation are stored in the test storage file. Moreover, users can use this mode to report bugs – the user can record the failed test case and forward it together with the description of error to the developer. As a rule, test cases are made according to the developer's interpretation of software specification. In the course of time, the amount and content of test cases increases due to system's evolution.

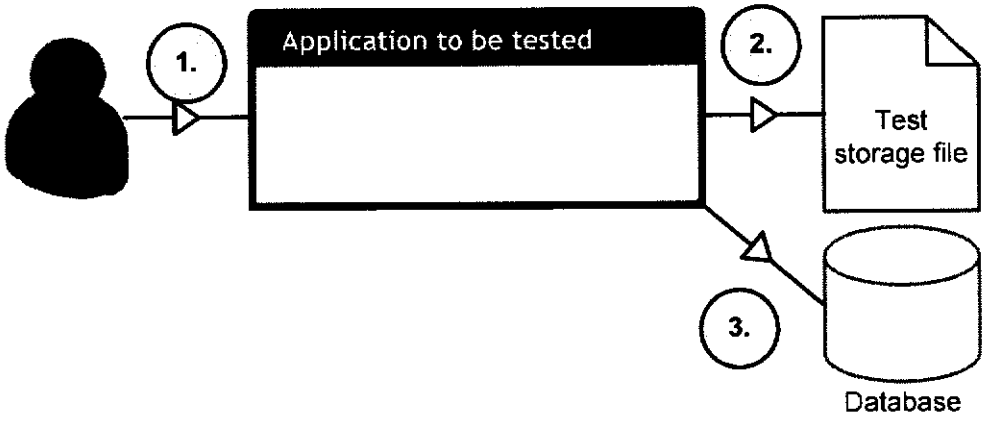

Fig. 1. Test Storage Mode

1. The user registers a test case in the test storage mode. The user uses the same application that is used for daily business purposes.
2. In the test storage mode, the application registers in the test file the actions performed in the system.
3. If data storing is performed in the application, the data are stored in the database. In the case of production environment, the related data are read from the real data-

base, while the storing is done in the test registration and execution database, not in the real database.

- 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. In the development and testing environments, test cases are executed in one database. In the production environment, the test storage and execution database is used. In the self-testing mode, the real database of the production environment is accessible in the read-only mode

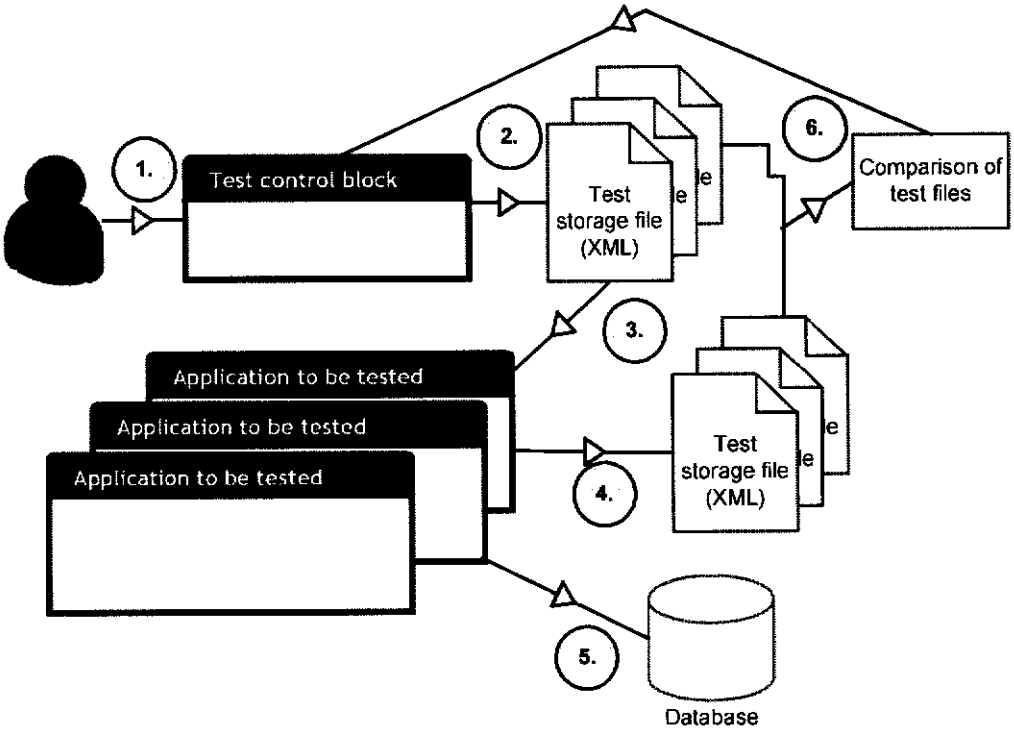

Fig. 2. Self-Testing Mode

1. To call system self-test, the user opens the test control block window.
2. The user loads the list of the available test files and selects the tests files to be executed.
3. The test control block reads information from the test file and executes it; the actions specified in the test file are performed.
4. In the self-testing mode, similarly to the test storage mode, a test file is created. The test file is created using the same approach as in the test storage mode.
5. If data storing is performed in the test case, the data are stored in the database. In the case of production environment, the related data are taken from the real database, while the storing is done in the test registration and execution database, not in the real database.
6. After the testing, the file created in the test storage mode is compared with the file created in the self-testing mode. If the contents of the files match, the test case has

been successful; if they do not match, the test case has failed. Information about test results is displayed in the user's test control block, where, if the test case has failed, the user can find detailed information about the reasons of the failure.

- 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 test cases in test storage files. During demonstration mode, form fields are automatically filled with test data from the test storage file, thus demonstrating the functionality of the system. Since test cases are taken from the test storage file, the demonstrator may rely that the demonstration will be always successful, avoiding any inconveniences and errors during the presentation. The demonstration mode process corresponds with the self-testing process (Fig. 2. Self-Testing Mode). The difference between the processes is that self-testing is performed in a mode invisible to the user, whereas the demonstration is performed step by step in a mode visible to the user. Together with the demonstration, it is possible to perform system self-testing. This approach is much slower, but it is possible to identify errors in a visible mode, executing the particular action that has been read from the test file.

3 Preparing the Database for Re-testing

The state of database during the test is crucial for the execution of test. It is possible that the state of database during the executions will differ from the state during the test storing. It means that there are cases where the test might, without reason, show a failure due to changes in the state of the database. For example: a test during which a certain amount is debited from the client's account is registered. If the test is performed repeatedly, it is possible that due to changes in account balance the test results will show a failure. Due to constantly changing database it is difficult and time-consuming to ensure that the stored test is executed with the same state of the database as of test registration moment. Solutions for executing re-tests, considering the variability of database, are described below:

- Creating a backup database. Prior to storing every test, a backup database is prepared, and a backup database is installed prior to executing every test. The backup is stored for as long as the stored tests, which have been registered using the particular backup, are being employed. This approach requires a lot of time and work resources;
- Generation of reverse tests. For every registered test case the self-testing software automatically generates a reverse test that reverses the database to the initial state. For example: If the user registers a test during which a certain amount is debited from the client's account, the self-testing software automatically generates a test that credits the particular amount to the client's account;
- Registration of reverse tests. The aim of the solution is to manage the registration of user tests so that they would contain a full set of events. The self-testing soft-

ware controls actions of the user who registers the test case, making the user to provide, with initial test cases, a data set with which the user later registers other test cases;

- Consecutive execution of all tests. When the self-testing functionality is implemented in the system, the database backup is taken. There after, a new backup is not taken from the database. Every time when system self-testing is performed, the taken database backup is installed and all the registered tests are performed consecutively;
- Priorities of self-testing mode. Priorities can be selected in the self-testing mode. If Priority 1 is selected, only those tests that are not dependant of the state of the database are executed. Self-testing mode with Priority 1 would be used by system users. If priority 2 is selected, all the stored tests will be executed. This priority will be used by system developers to perform testing, and they will prepare a database prior to testing according to the test requirements;
- Test execution criteria. Criteria of successful test execution are built in the system. From the solutions described above, the test execution criteria approach will be used in the system self-testing. Key considerations for the use of this approach are outlined in the next Section.

3.1 Test execution criteria

To ensure that tests are not dependant of the data set on which the tests are performed, it is necessary to control the key criteria for successful execution of the test and without which the execution of the test is impossible. The control is ensured with test execution criteria built in the system under test, which during the test check that it is possible to execute the specified criterion. If the criterion specified in the test point does perform, the test continues to execute; if not, the test execution is terminated and is marked as failed. The reason of termination is notified to the user, who can eliminate the failure and execute the test again. For example: a test case where a certain amount is debited from the client's account. In this test, the following execution criteria could be implemented and controlled:

- The amount specified in the test case is available in the client's bank account;
- The bank account specified in the test case is registered in the name of the client;
- The client's bank account is not closed.

Advantages of this approach:

- When the test is being executed, it is not necessary to provide the same state of the database it was in when registering the test;
- The solution is not time-consuming. To execute tests, a database backup need not be installed;
- It is possible to execute a particular test(s) not executing all the registered tests;
- The technical implementation is comparatively simple.

The major drawback of this solution is the extra work to be done to implement the test execution criteria in the software.

Hereinafter the paper deals with the Test Point concept. Test execution criteria in the self-testing software are realised as test points, which are in line with the general approach for realisation of self-testing.

4 Test Point

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.

To show how test points are used, a stock purchase transaction process is shown in the figure below (Fig. 3. 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.

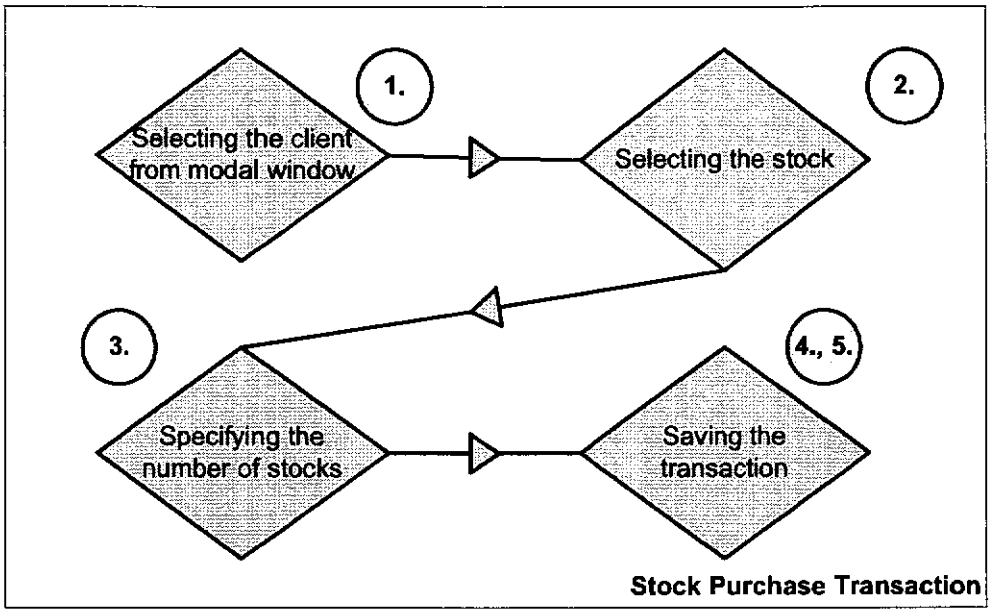

Fig. 3. Stock Purchase Transaction Process

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 *Modal window* registers the client selected in it in the test storage file.
2. Test point *Field with value* registers in the test storage file the security specified for the transaction.
3. Test point *Field with value* registers in the test storage file the quantity of securities specified for the transaction.
4. Test point *Application event* registers in the test storage file the event of clicking on the button Save.
5. Test point *SQL query result* registers in the test storage file the data saved in the database after clicking on the button Save.

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.

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. Possible types of test points and their use is described further in this Section;
- 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.

Self-testing software employs the following testing actions, the use of which is shown in Table 1:

- Field with value. The action is required to register a field filling-in event;
- Comparable value. This test point is necessary to be able to register and compare 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;
- MessageBox. This test point is required to be able to simulate the message box action, not actually calling the messages. This is necessary as not all technologies provide a possibility to press the message button with the help of the system during test execution;
- Modal window. This test point is required to be able to simulate the modal window action, not actually calling the modal windows. This is necessary as not all technologies provide a possibility to access during test execution, after calling the modal window, the window from which the modal window is called;
- SQL query result. 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;
- Application event. This test point is required to register any events performed in the application, e.g. clicking on the button Save;
- Test execution criterion. This test point controls whether it is possible to execute the test. 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.

Table 1. Types of Testing Actions

Test storage	**Self-testing**	**Demonstration**
Field with value		
Registering the field name and field value in the test file.	Reading the field name and field value from the test file and writing the value in the respective field.	See text execution mode.
Comparable value		
Registering various values, inter alia values obtained from the calculation, in the test file.	There are two cases (additional attribute that points to the particular case): • Comparing the value calculated during test execution with the value registered in the test file during test storage; • Using the value registered in the test file during test storage in test execution, not recalculating the value.	Using the value registered in the test file during test storage in test execution, not recalculating the value.
Message box		
Registering in the test file the message call and the action performed by the user.	The message is not shown to the user. Tests are executed taking into account the action performed by the user and registered in the test file.	In this mode, the message box is shown on the screen.
Modal window		
Registering in the test file the modal window call and the return values.	The modal window is not opened during test execution, and the modal window values registered in the storage mode are used from the test file.	In this mode, the modal window is shown on the screen.
SQL query result		
Registering the test results in the test file. Test results can be both a particular field value and a query result.	Comparing the values obtained in the result of test execution with the values registered in the test file during test storage mode.	Test points are not used in this mode.
Application event		
Registering application events in the test file.	Reading application events from the test file and executing them.	See text execution mode.
Test execution criterion		
The action is not used in this mode.	Controlling whether it is possible to execute the tests in the current state of database.	See text execution mode.

5 Implementation of Self-Testing

Key components of self-testing software are:

1. Test control block, which provides the following key functions:

- Selecting the test execution mode (execution of all or selected tests);
- 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;
- 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.

As of writing this paper, the first version of the test control block has been developed (Fig. 4. Test Control Block). The test control block has been developed with additional functionality and improved with user interface.

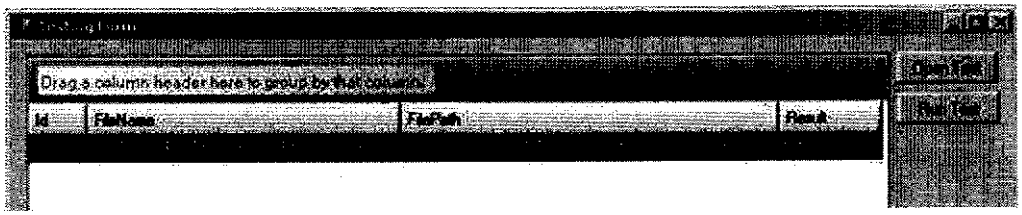

Fig. 4. Test Control Block

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. 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 consists of the following elements and their attributes:

- Form. Its attributes are the form name and the test point number, and its elements are test points;
- Action. The element is a set of other elements. It contains all the test points described in Section 4;
- Control. The element contains data on the control used in the test point. The element contains the following attributes: test point number, control name, event (e.g. change of value) called at the test point, control type;
- Value. Value element. Contains information on the value selected/entered by the control. In addition, the element can contain the value data type (e.g.: xsi:type="xsd:string" – which means that the control value is a string of symbols);
- Values. Element values. If the control contains a number of selected values, they are shown under this element. The element contains the Value element;

- Function. This test point determines whether a function has been called. The element contains the Parameters element and the following attributes: test point number, function name;
- Parameter. Function parameter. Contains information on the value of the parameter of the called function. In addition, the element can contain the value data type;
- Parameters. The element is a set of function parameters. The element can contain the Parameter elements;
- ModalFields. The element contains information on the return values of the modal window. The element contains the Fields element and the test point number attribute;
- Field. Modal window return value element;
- Query. The element contains the SQL query, which is executed by the system in the database. Its attribute is the test point number;
- ComparableField. The element contains information on the field that must be registered when the test is recorded in order to be able, when the test is played back, to check whether the value matches the value that was registered when the test was recorded. The element contains the following attributes: test point number, field name. The element contains the Value element, in which the field value is specified;
- DialogResult. The element contains information on the return values of the dialog box. The element contains the following attributes: test point number, dialog result;
- ChildForm. ChildForm matches with the Form element (form child). The element is required if another form is called from the form.

The system login test example, in which test points are located, is shown in the figure below (Fig. 5. System Login Test Example).

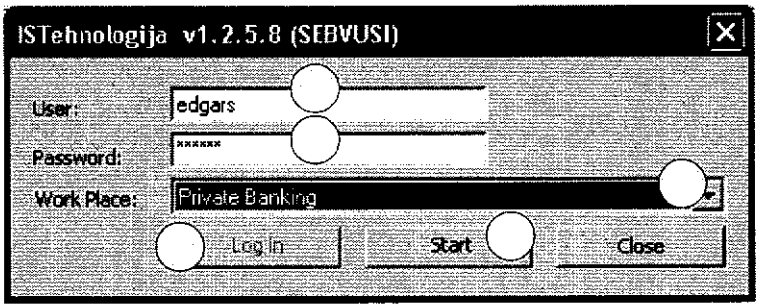

Fig. 5. System Login Test Example

The user performs the following actions in login form:

1. Entering the user name.
2. Entering the user password.
3. Clicking on the button Log in.
4. Choosing work place.
5. Clicking on the button Start.

When a system login test case is registered, each of the points in the test storage file registers information (Fig. 6. Test File Example) that is used to play back the test.

```
- <Actions>
  - <Control Id="1" Name="Username" Event="Username_TextChanged"
      ControlType="System.Windows.Forms.TextBox">
      <Value xsi:type="xsd:string">edgars</Value>
    </Control>
  - <Control Id="2" Name="Passwd" Event="Passwd_TextChanged"
      ControlType="System.Windows.Forms.TextBox">
      <Value xsi:type="xsd:string">123456</Value>
    </Control>
    <Control Id="3" Name="btnAuth" Event="btnAuth_Click"
      ControlType="System.Windows.Forms.Button" />
  - <Query Id="4">
      <Query>SELECT d.DVI_NOSAUK,d.DVI_ISN FROM IST_DVI d, IST_SLD s
        WHERE s.DVI_ISN=d.DVI_ISN AND s.DAR_ISN=72 order by 1</Query>
    </Query>
  - <Control Id="5" Name="Workplace"
      Event="Workplace_SelectedIndexChanged"
      ControlType="System.Windows.Forms.ComboBox">
      <Value xsi:type="xsd:int">20</Value>
    </Control>
    <Control Id="6" Name="StartWork" Event="StartWork_Click"
      ControlType="System.Windows.Forms.Button" />
</Actions>
```

Fig. 6. Test File Example

In the example (Fig. 5. System Login Test Example) the order in which test is recorded (1-2-3-4-5) is not fixed. The user in login form could perform actions in any order (for example 2-1-3-4-5). The order in which test case will executed will be the same as the sequence in which the test is recorded.

The test functions library described above can be used in projects developed in the MS Visual Studio environment. If required, it can be easily supplemented with new functions.

6 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.

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: Accepted for publication in the 4th IFIP TC2 Central and East European Conference on Software Engineering Techniques (CEE-SET 2009), Krakow, Poland, Oktober 12-14, 2009
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)