SCIENTIFIC PAPERS, UNIVERSITY OF LATVIA, 2012. Vol. 787
COMPUTER SCIENCE AND INFORMATION TECHNOLOGIES 7–26 P.

Efficiency Measurements of Self-Testing

Edgars Diebelis

University of Latvia, Raina blvd. 19, Riga, Latvia, *edgars.diebelis@di.lv*

This paper is devoted to efficiency measurements of self-testing, which is one of smart technology components. The efficiency measurements were based on the statistics on incidents registered for eight years in a particular software development project: Currency and Securities Accounting System. Incident notification categories have been distributed into groups in the paper: bugs that would be identified, with appropriately located test points, already in the development environment, and bugs that would not be identified with the self-testing approach neither in the development, testing or production environments. The real measurements of the self-testing approach provided in the paper prove its efficiency and expediency.

Keywords. Testing, Smart technologies, Self-testing.

Introduction

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, for examples, external environment testing [2, 3], intelligent version updating [4], integration of the business model in the software [5] and others. The necessity of this feature is driven by the growing complexity of information systems. The concept of smart technologies is aiming at similar goals as the concept of autonomous systems developed by IBM in 2001 [6, 7, 8], but is different as for the problems to be solved and the solution methods. Autonomic Computing purpose [9] is to develop information systems that are able of self-management, in this way taking down the barriers between users and the more and more complex world of information technologies. IBM outlined four key features that characterise autonomic computing:

- Self-configuration. Automated configuration of components and systems follows high-level policies. Rest of system adjusts automatically and seamlessly.
- Self-optimization. Components and systems continually seek opportunities to improve their own performance and efficiency.
- Self-healing. System automatically detects, diagnoses, and repairs localized software and hardware problems.

- Self-protection. System automatically defends against malicious attacks or cascading failures. It uses early warning to anticipate and prevent system-wide failures.

With an aim similar to an autonomic computing, the smart technologies approach was offered in 2007 [1]. It provided a number of practically implementable improvements to the functionality of information systems that would make their maintenance and daily use simpler, taking us nearer to the main aim of autonomic computing. The main method offered for implementing smart technologies is not the development of autonomous universal components but the direct implementation of smart technologies in particular programs.

The first results of practical implementation of smart technologies are available. Intelligent version updating software is used in the Financial and budget report (FIBU) information system that manages budget planning and performance control in more than 400 government and local government organisations with more than 2000 users [4]. External environment testing [3] is used in FIBU, the Bank of Latvia and some commercial banks (VAS Hipotēku un zemes banka, AS SEB banka etc.) in Latvia. 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 Software Development (MDSD) [10], and it is used in developing and maintaining several event-oriented systems.

Self-testing provides the software with a feature to test itself automatically prior to operation. 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 article continues to deal with the analysis of the self-testing approach. Previous papers [11, 13, 14] have dealt with the ideology and implementation of the self-testing approach. 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).

In order to ensure development of high quality software, it is recommendable to perform testing in three phases in different environments: development, test, production [11]. Self-testing implementation is based on the concept of Test Points [13, 14]. 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. Key components of the self-testing software are: test control block for capturing and playback of tests, library of test actions, test file (XML file) [13, 14].

This paper analyses the efficiency of self-testing. Since it is practically impossible to test a particular system with various methods and compare them (e.g. manual

tests, tests using testing support tools with various features) with tests performed using the self-testing approach, another methodology for evaluating the efficiency has been used. The implementation of a comparatively large system was selected, and an expert analysed what bugs have been detected and in what stage of system development they could be detected, assuming that the self-testing functionality in the system would had been implemented from the very beginning of developing it. The analysis was based on the statistics of incident notifications registered during the system development stage, in which testing was done manually. The paper contains tables of statistics and graphs distributed by types of incident notifications, types of test points and years; this makes it possible to make an overall assessment of the efficiency of self-testing.

The paper is composed as follows: Chapter 1 briefly outlines the Currency and Securities Accounting System in order to give an insight on the system whose incident notifications were used for measuring the efficiency of the self-testing approach. Chapter 2 deals in more detail with the approach used for measuring the efficiency of self-testing. Chapter 3 provides a detailed overview of the measurement results from various aspects, and this makes it possible to draw conclusions on the pros and cons of self-testing.

1. Currency and Securities Accounting System (CSAS)

Since the paper's author has managed, for more than six years, the maintenance and improvement of the Currency and Securities Accounting System (CSAS), it was chosen as the test system for evaluating the self-testing approach. The CSAS, in various configuration versions, is being used by three Latvian banks:

- SEB banka (from 1996);
- AS Reģionālā investīciju banka (from 2007);
- VAS Latvijas Hipotēku un zemes banka (from 2008),

The CSAS is a system built in two-level architecture (client-server), and it consists of:

- client's applications (more than 200 forms) developed in: Centura SQL Windows, MS Visual Studio C#, MS Visual Studio VB.Net;
- database management system Oracle 10g (317 tables, 50 procedures, 52 triggers, 112 functions).

The CSAS consists of two connected modules:

- securities accounting module (SAM);
- currency transactions accounting module (CTAM).

1.1. Securities Accounting Module (SAM)

The software's purpose is to ensure the execution and analysis of transactions in securities. The SAM makes it possible for the bank to register customers that

hold securities accounts, securities and perform transactions in them in the name of both the bank and the customers. Apart from inputting and storing the data, the system also makes it possible to create analytical reports.

The SAM ensures the accounting of securities held by the bank and its customers and transactions in them. Key functions of the SAM are:

- Inputting transaction applications. The SAM ensures the accounting of the following transactions: selling/buying securities, security transfers, transactions in fund units, securities transfers between correspondent accounts, deregistration of securities, repo and reverse repo transactions, encumbrances on securities, trade in options;
- Control of transaction application statuses and processing of transactions in accordance with the determined scenario;
- Information exchange with external systems. The SAM ensures information exchange with the following external systems: the bank's internet banking facility, fund unit issuer's system, the Latvian Central Depository, the Riga Stock Exchange, the bank's central system, Bloomberg, SWIFT, the Asset Management System (AMS);
- Registration and processing of executed transactions;
- Calculation and collection of various fees (broker, holding etc);
- Revaluation of the bank's securities portfolio, amortisation and calculation of provisions;
- Control of partner limits;
- Comparing the bank's correspondent account balances with the account statements sent by the correspondent banks;
- Making and processing money payment orders related to securities. The SAM ensures the accounting of the following payments: payment of dividends (also for deregistered shares), tax collection/return, coupon payments, principal amount payments upon maturity;
- Making reports (inter alia to the Financial and Capital Market Commission, the Bank of Latvia, securities account statements to customers).

The SAM ensures swift recalculation of securities account balances based on transactions. The SAM is provided with an adjustable workplace configuration.

1.2. Currency Transactions Accounting Module (CTAM)

The purpose of the software is to account and analyse currency transactions (currency exchange, interbank deposits etc). The system accounts currency transactions and customers who perform currency transactions. Apart from data accounting, the system provides its users with analytical information on customers, currency transactions and currency exchange rates.

Key functions of the CTAM are:

- Inputting transactions. The CTAM ensures the accounting of the following transactions: interbank FX, FX Forward, SWAP (risk, risk-free), interbank depositing, (in/out/extension); customer FX, FX Forward, SWAP (risk, risk-free), interbank depositing, (in/out/extension), floating rate, interbank order, customer order, interest rate swaps, options, State Treasury transactions, collateral transactions, currency swap transactions;
- Control of transaction status and processing of transactions in accordance with the determined scenario (workflow);
- Information exchange with external systems. The CTAM ensures information exchange with the following external systems: bank's central system, Reuter, UBS TS trading platform, SWIFT;
- Maintaining currency positions (bank's total position, positions per portfolios);
- Setting and controlling limits;
- Importing transactions from Reuter and Internet trading platforms (TS, UBS TS etc);
- Importing currency exchange rates and interest rates from Reuters;
- Margin trading;
- Nostro account balances;
- Making reports.

The CTAM is provided with an adjustable workplace configuration.

2. Self-testing Efficiency Measurements Approach

All incident notifications (1,171 in total) in the CSAS in the period from July 2003 to 23 August 2011 retrieved from the Bugzilla [15] incident logging system were analysed. Since the incident logging system is integrated with a work time accounting system, in the efficiency measurements not only the quantity of incidents registered but also the time consumed to resolve incidents was used. Every incident notification was evaluated using the criteria provided in Table 1. As it can be seen in the table, not all incident notifications have been classified as bugs. Users register in the incident logging system also incidents that, after investigating the bug, in some cases get reclassified as user errors, improvements or consultations.

Table 1

Types of Incident Notifications

Type of Incident	Description
Unidentifiable bug	Incident notifications that described an actual bug in the system and that would not be identified by the self-testing approach if the system had a self-testing approach tool implemented.
Identifiable bug	Incident notifications that described an actual bug in the system and that would be identified by the self-testing approach if the system had a self-testing approach tool implemented.
Duplicate	Incident notifications that repeated an open incident notification
User error	Incident notifications about which, during resolving, it was established that the situation described had occurred due to the user's actions.
Improvement	Incident notifications that turned out to be system functionality improvements.
Consultation	Incident notifications that were resolved by way of consultations.

For measuring the efficiency of the self-testing approach, the most important types of incident notifications are Identifiable Bug and Unidentifiable Bug. Therefore, these types of incident notifications are looked at in more detail on the basis of the distribution of incidents by types provided in the tables below (Table 2 and Table 3).

Table 2

Unidentifiable Bug; Distribution of Notifications by Bug Types

Bug type	Description
External interface bug	Error in data exchange with the external system
Computer configuration bug	Incompliance of user computer's configuration with the requirements of the CSAS.
Data type bug	Inconsistent, incorrect data type used. Mismatch between form fields and data base table fields
User interface bug	Visual changes. For example, a field is inactive in the form or a logo is not displayed in the report.
Simultaneous users actions bug	Simultaneous actions by multiple users with one record in the system.
Requirement interpretation bug	Incomplete customer's requirements. Erroneous interpretation of requirements by the developer.
Specific event	Specific uses of the system resulting in a system error.

Table 3

Identifiable Bug; Distribution of Notifications by Test Point Types

Test point that would identify the bug	**Description**
File result test point	The test point provides the registration of the values to be read from and written to the file, inter alia creation of the file.
Input field test point	This test point registers an event of filling in a field.
Application event test point	This test point would register any events performed in the application, e.g. clicking on the Save button;
Comparable value test point	This test point registers 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, the values of which are not saved in the database.
System message test point	This test point is required to be able to simulate the message box action, not actually calling the messages.
SQL query result test point	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 test storing and registered in the test file.

3. Results of Measurements

3.1. Distribution of Notifications by Incident Types and Time Consumed

Table 4 shows a distribution of all incident notifications by incident types and the time consumed to resolve them in hours. The table contains the following columns:

- Notification type – incident notification type (Table 1);
- Quantity – quantity of incident notifications
- % of total – percentage of a particular notification type in the total quantity of incident notifications;
- Hours – total time consumed to resolve one type of incident notifications;
- % of total – percentage of the time spent to resolve the particular notification type of the total time spent for resolving all incident notifications.

Table 4

Distribution of All Notifications by Incident Types and the Time Consumed to Resolve Them

Application type	Quantity	% of total	Hours	% of total
Duplicate	68	5.81	23.16	0.47
User error	43	3.67	67.46	1.37
Unidentifiable bug	178	15.2	1,011.96	20.52
Identifiable bug	736	62.85	3,293.74	66.79
Improvement	102	8.71	241.36	4.89
Consultation	44	3.76	293.92	5.96
Total:	**1,171**	**100**	**4,931.6**	**100**

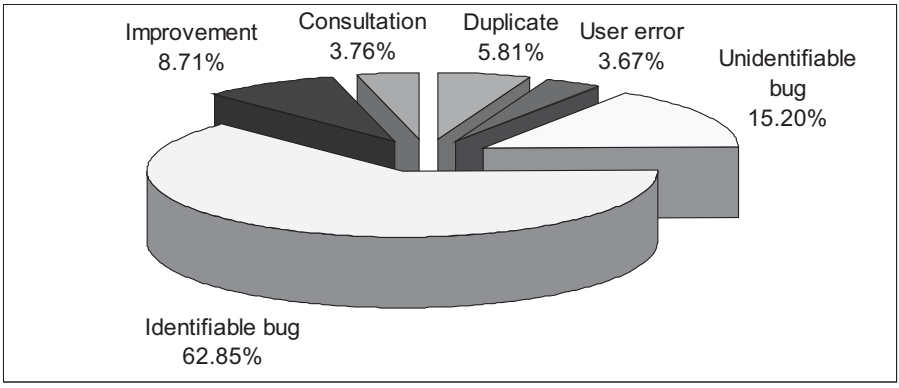

Figure 1. Distribution of All Notifications by Incident Types

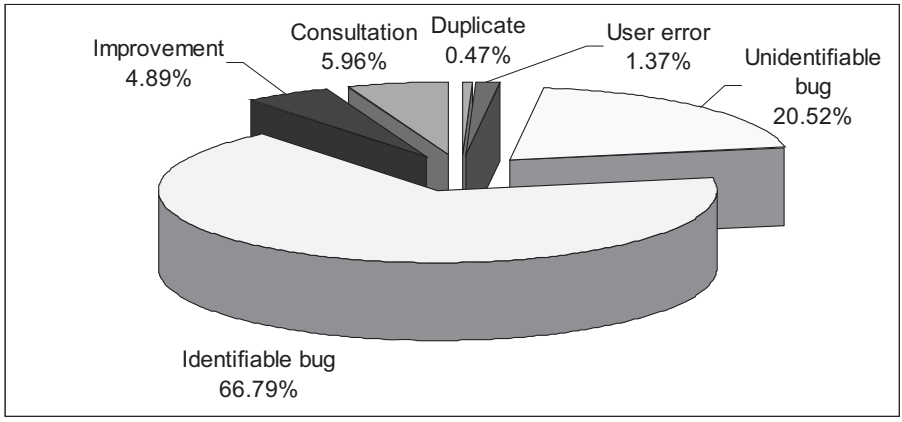

Figure 2. Distribution of All Notifications by the Time Consumed to Resolve per Incident Type

Considering the information obtained, the author concludes that:

- Of the total quantity of incident notifications (1,171), if the self-testing approach had been implemented in the CSAS, 62.85% (736 notifications, 3,293.74 hours consumed for resolving) of all the incident notifications registered in nearly nine years would had been identified by it already in the development stage. It means that developers would had been able to identify and repair the bugs already in the development stage, which would significantly improve the system's quality and increase the customer's trust about the system's quality. Measurements similar to [16] show the advantages of systematic testing compared to manual.
- As mentioned in many sources [17], the sooner a bug is identified, the lower the costs of repairing it. The differences mentioned reach to ten and even hundred times. The time spent for repairing bugs as provided in Table 4 would be definitely lower if the bugs had been identified already in the development stage. Assuming that the identification of bugs in the development stage would allow saving 50% of the time consumed for repairing all the bugs identified by the customer, about 1,650 hrs, or about 206 working days, could have been saved in the period of nine years.
- Of the total quantity of incident notifications, the self-testing approach in the CSAS would not be able to identify 15.2% of the bugs (178 notifications, 1,011.74 hours consumed for repairing) of the total number of incident notifications.
- Of the total quantity of incident notifications, 78.05% (914 notifications, 4,305.7 hours consumed for repairing) were actual bugs that were repaired, other 11.95% (257 notifications, time consumed for repairing 625.9 hours) of incident notifications were user errors, improvements, consultations and bug duplicates that cannot not be identified with testing.
- The time consumed for repairing bugs in percentage (87.31%) of the total time consumed for incident notifications is higher than the percentage (78.05%) of bugs in the total quantity of incident notifications. This means that more time has been spent to repair bugs proportionally to other incident notifications (improvements, user errors, consultations to users and bug duplicates).

3.2. Distribution of Notifications by Bug Types

As mentioned in the previous Chapter, there are two types of incident notifications that are classified as bugs: Unidentifiable Bugs and Identifiable Bugs, and they are analysed in the next table (Table 5). The table columns' descriptions are the same as in the previous Chapter.

Table 5

Distribution of Bugs by Bug Types

Bug Type	Quantity	% of total	Hours	% of total
Unidentifiable bug	178	19.47	1011.96	23.5
Identifiable bug	736	80.53	3293.74	76.5
Total:	**914**	**100**	**4305.7**	**100**

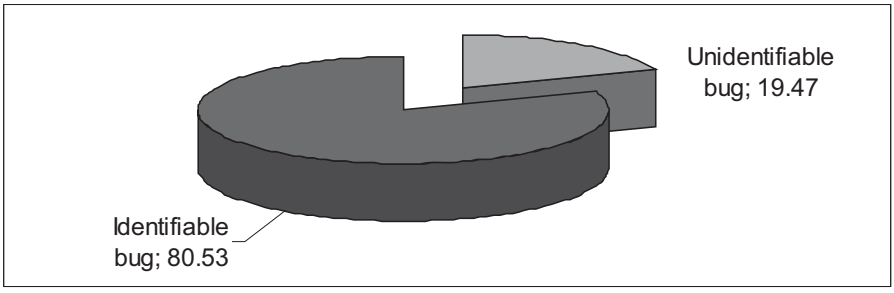

Figure 3. Distribution of Bugs by Bug Types

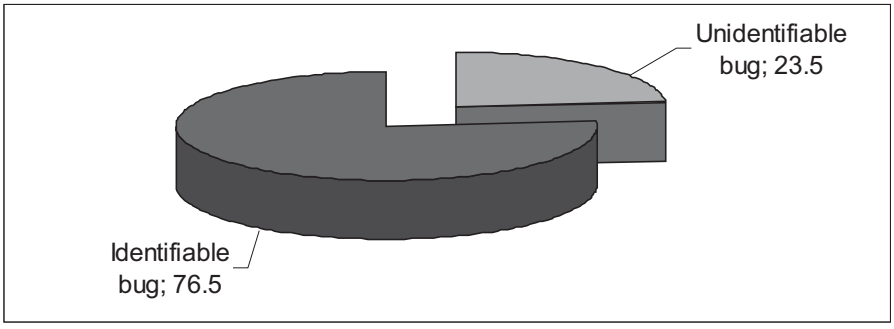

Figure 4. Distribution of Bugs by Bug Types per Time Consumed

Considering the information obtained, the author concludes that:

- The self-testing approach would identify 80% of all the bugs registered in the CSAS.
- The time consumed for repairing the bugs identified by the self-testing approach in percent (76.5%) of the total time consumed for resolving is lower than the percentage (80.53%) of these bugs in the total quantity of bugs. This means that less time would be spent to repair the bugs identified with the self-testing approach proportionally to the bugs that would not be identified with self-testing.

3.3. Distribution of Notifications by Years

The next table (Table 6) shows the distribution of incident notifications by years. The table contains the following columns:

- Notification type
  - quantity of incident notifications by type (Table 1) per years;
  - % of total – percentage of a particular notification type in the total quantity of incident notifications per year;
- 2003-2011 – years analysed;

Table 6

Distribution of Notifications by Years

Notification type	**2003**	**2004**	**2005**	**2006**	**2007**	**2008**	**2009**	**2010**	**2011**
Duplicate	1	1	13	22	7	10	7	5	2
% of total	3.23	1.3	13.13	10.19	4.9	5.29	3.45	3.85	2.41
User error	2	6	3	10	1	7	5	3	6
% of total	6.45	7.79	3.03	4.63	0.7	3.7	2.46	2.31	7.23
Unidentifiable bug	2	7	11	19	19	37	38	23	22
% of total	6.45	9.09	11.11	8.8	13.29	19.58	18.72	17.69	26.51
Identifiable bug	17	51	59	138	98	110	141	79	43
% of total	54.84	66.23	59.6	63.89	68.53	58.2	69.46	60.77	51.81
Improvement	9	12	13	25	11	11	6	11	4
% of total	29.03	15.58	13.13	11.57	7.69	5.82	2.96	8.46	4.82
Consultation	0	0	0	2	7	14	6	9	6
% of total	0	0	0	0.93	4.9	7.41	2.96	6.92	7.23
Total:	**31**	**77**	**99**	**216**	**143**	**189**	**203**	**130**	**83**

From the table it can be seen that:

- In some years, there are peaks and falls in the number of some incident notification types; also, consultation-type incident notifications have been registered as from 2005, but their proportional distribution by years match approximately the total proportional distribution of notifications.
- In any of the year's most of the bugs notified could be identified with the self-testing approach.

3.4. Distribution of Bugs by Years

The next table (Table 7) shows the distribution of bugs separately. The table contains the following columns:

- Year – years (2003-2011) analysed;

- Udentifiable bug – number of udentifiable bugs by years;
- % of total – quantity of udentifiable bugs as a percentage of the total number of udentifiable bugs;
- Identifiable bug – number of identifiable bugs by years;
- % of total – quantity of identifiable bugs as a percentage of the total number of identifiable bugs;
- Total – totals of bug quantities.

Table 7

Distribution of Bugs by Years

Year	Unidentifiable bug	% of total	Identifiable bug	% of total	Total
2003	2	1.12	17	2.31	**19**
2004	7	3.93	51	6.93	**58**
2005	11	6.18	59	8.02	**70**
2006	19	10.67	138	18.75	**157**
2007	19	10.67	98	13.32	**117**
2008	37	20.79	110	14.95	**147**
2009	38	21.35	141	19.16	**179**
2010	23	12.92	79	10.73	**102**
2011	22	12.36	43	5.84	**65**
Total	**178**	**100**	**736**	**100**	**914**

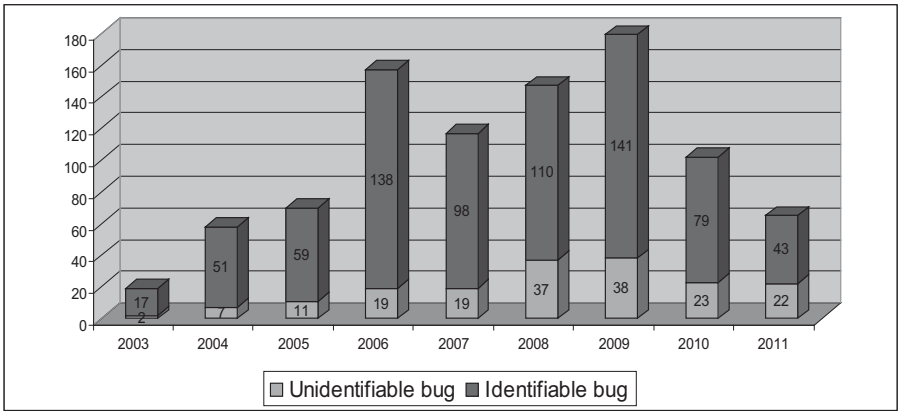

Figure 5. Distribution of Bugs by Years

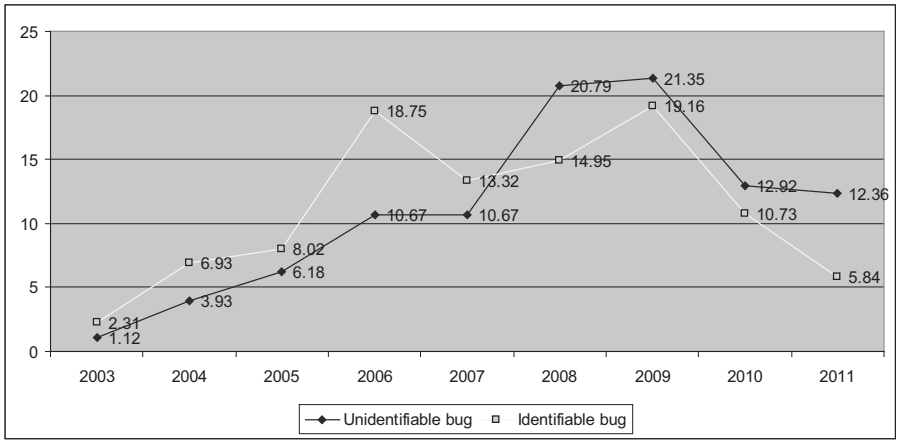

Figure 6. Distribution of Bugs in Percent by Years

Considering the information obtained, the author concludes that:

- The self-testing approach would identify most of the bugs registered in the CSAS.
- Changes, by years, in the percentages of the bugs identified and not identified with the self-testing approach are not significant.
- In the first five years, the weighting of the bugs that could be identified with the self-testing approach is higher, but later the weighting of the bugs that could not be identified with the self-testing approach becomes higher. This shows that the "simpler" bugs are discovered sooner than the "non-standard" ones.

3.5. Ratio of the Bug Volume to the Improvement Expenses Distributed by Years

The next table (Table 9) shows the ratio of the quantity of bugs to the volume of improvement expenses by years. The table contains the following columns:

- Year – years analysed;
- Quantity of bugs – quantity of bugs registered in the CSAS by years;
- % of total – distribution of the quantity of bugs in percent by years;
- Improvement expenses in % of total – percentage of the amount spent for improvements by years;

Table 8

Ratio of the Bug Quantity to the Improvement Expenses Distributed by Years

Year	Quantity of bugs	% to total	Improvement expenses in % of total
2003	31	2.65	2.86
2004	77	6.58	12.38
2005	99	8.45	15.25
2006	216	18.45	15.13
2007	143	12.21	11.04
2008	189	16.14	21.62
2009	203	17.34	9.77
2010	130	11.1	6.24
2011	83	7.09	5.71
Total:	**1,171**	**100**	**100**

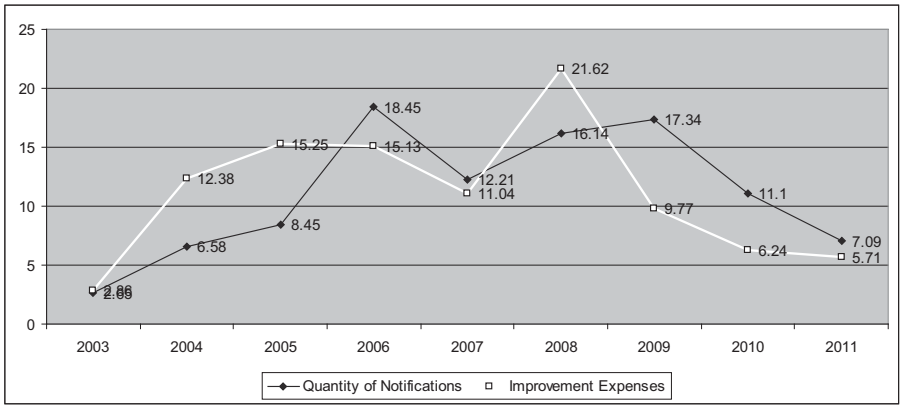

Figure 7. Ratio of the Quantity of Notifications to the Improvement Expenses

The information looked at in this Sub-chapter does not directly reflect the efficiency of the self-testing approach, but it is interesting to compare changes in the quantity of notifications (%) and in the improvement expenses (%) during the nine years. Considering the information obtained, it can be concluded that, as the improvement volumes grow, also the bug volumes grow. The conclusion is a rather logical one, but in this case it is based on an actual example.

3.6. Distribution of the Bugs Unidentifiable by the Self-Testing Approach by Types

The next table (Table 9) shows the distribution of bugs unidentifiable by the self-testing approach by bug types. The table contains the following columns:

- Bug type – one of seven bug types;
- Quantity – quantity of bugs of the type;
- % of total – percentage of the bug type in the total quantity of bugs.

Table 9

Distribution of the Bugs Unidentifiable by the Self-Testing Approach by Types

Bug type	**Quantity**	**% of Total**
External interface bug	5	2.81
Computer configuration bug	12	6.74
Data type bug	7	3.93
User interface bug	25	14.04
Simultaneous actions by users	5	2.81
Requirement interpretation bug	41	23.03
Specific event	83	46.63
Total:	**178**	**100**

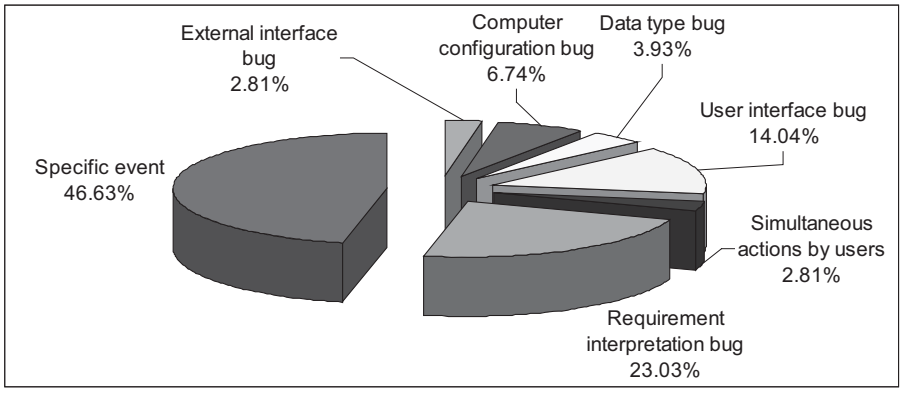

Figure 8. Distribution of the Bugs Unidentifiable by the Self-Testing Approach by Types

Conclusions:

- Most (nearly 50%) of the bugs that the self-testing approach would not be able to identify are specific cases that had not been considered when developing the system. For example, the following scenario from a bug description: "I enter the login, password, press Enter, press Enter once again, then I press Start Work, in the tree I select any view that has items under it. And I get an error!". As it can be seen, it is a specific case that the

developer had not considered. To test critical functionality, a test example that plans that Enter is pressed once would be made, and this test example would not result in a bug. Furthermore, a scenario that plans that Enter is pressed twice would not be created since it is a specific case not performed by users in their daily work. The self-testing approach is not able to identify bugs that occur due to various external devices. For example, when a transaction confirmation is printed, the self-testing approach would not be able to detect that one excessive empty page will be printed with it.

- One fifth of the total quantity of the bugs that the self-testing approach would be unable to identify are requirement interpretation bugs. Bugs of this type occur when system additions/improvements are developed and the result does not comply with the customer's requirements because the developer had interpreted the customer's requirements differently. To the author's mind, the quantity of hours (41) during the nine-year period is small and is permissible.
- The self-testing approach is unable to identify visual changes in user interfaces, data formats, field accessibility and similar bugs.
- A part of the registered bugs are related to incompliance of the user computer's configuration with the system requirements. The self-testing approach would be able to identify bugs of this type only on the user computer, not on the testing computer that has been configured in compliance with the system requirements.
- A part of the bugs that the self-testing approach would be unable to identify are data type bugs that include:
    - checking that the window field length and data base table field length match;
    - exceeding the maximum value of the variable data type.
- The self-testing approach is unable to identify bugs that result from the data of external interfaces with other systems. The self-testing approach is able to store and execute test examples that contain data from external interfaces, but it is unable to create test examples that are not compliant with the requirements of the external system (e.g. a string of characters instead of digits is given by the internal interface). Of course, it is possible to implement a control in the system itself that checks that the data received from the external interface are correct.
- The self-testing approach is unable to identify bugs that result from transaction mechanisms incorrectly implemented in the system, e.g. if several users can simultaneously modify one and the same data base record.

3.7. Distribution of the Bugs Identifiable by the Self-Testing Approach by Test Point Types

The next table (Table 10) shows the distribution of bugs identifiable by the self-testing approach by bug types and time consumed to resolve them. The table contains the following columns:

- Test point – test point that would identify the bug;
- Quantity – quantity of bugs that the test point would identify;
- % of total – percentage of the bugs that would be repaired by the test point in the total quantity of bugs that would be repaired by all test points;
- Hours – hours spent to resolve the bugs that the test points could identify;
- % of total – percentage of hours in the total number of hours consumed to repair the bugs that could be identified by test points.

Table 10

Distribution of the Bugs Identifiable by the Self-Testing Approach by Test Point Types

Test point	Quantity	% of total	Hours	% of total
File result test point	59	8.02	150.03	4.56
Entry field test point	146	19.84	827.14	25.11
Application event test point	105	14.27	364.24	11.06
Comparable value test point	28	3.8	93.53	2.84
System message test point	11	1.49	58.84	1.79
SQL query result test point	387	52.58	1,799.96	54.65
Total:	**736**	**100**	**3,293.74**	**100**

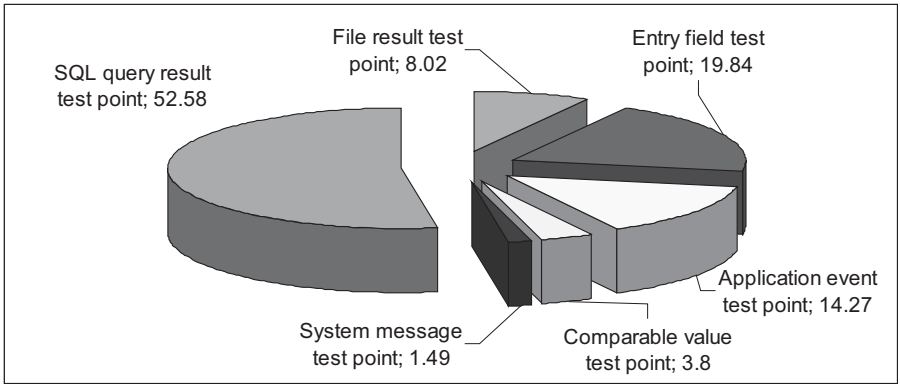

Figure 9. Distribution of the Bugs Identifiable by the Self-Testing Approach by Test Point Types

Conclusions:

- More than a half of all the registered bugs could be identified with the SQL query result test point. At the test point, data are selected from the data base and compared with the benchmark values. The explanation is that the key purpose of the CSAS is data storing and making reports using the stored data.
- One fifth of the bugs that could be identified with the self-testing approach would be identified by the input field test point. The test point compares the field value with the benchmark value.

4. Conclusions

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

- Using the self-testing approach, developers would have been able to identify and repair 80% of the bugs already in the development stage; accordingly, 63% of all the received incident notifications would have never occurred. This would significantly improve the system's quality and increase the customer's trust about the system's quality.
- A general truth is: the faster a bug is identified, the lower the costs of repairing it. The self-testing approach makes it possible to identify many bugs already in the development stage, and consequently the costs of repairing the bugs could be reduced, possibly, by two times.
- Most of the bugs that the self-testing approach would be unable to identify are specific cases of system use.
- The SQL query result test point has a significant role in the identification of bugs; in the system analysed herein, it would had identified more than a half of the bugs notified.

From the analysis of the statistics, it can be clearly concluded that the implementation of self-testing would make it possible to save time and improve the system quality significantly. Also, the analysis has shown that the self-testing approach is not able to identify all system errors. On the basis of the analysis provided herein, further work in evolving the self-testing approach will be aimed at reducing the scope of the types of bugs that the current self-testing approach is unable to identify.

IEGULDĪJUMS TAVĀ NĀKOTNĒ

This work has been supported by the European Social Fund within the project «Support for Doctoral Studies at University of Latvia».

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 Ç.: 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. Kephart, J., O., Chess, D., M. The Vision of Autonomic Computing. In: Computer Magazine, vol. 36, pp.41-50 (2003).
10. Barzdins, J., Zarins, A., Cerans, K., Grasmanis, M., Kalnins, A., Rencis, E., Lace, L., Liepins, R., Sprogis, A., Zarins, A.: 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.
14. Diebelis, E., Bicevskis, J.: Test Points in Self-Testing. In: Marite Kirikova, Janis Barzdins (eds.) Databases and Information Systems VI, Selected Papers from the Ninth International Baltic Conference. IOS Press vol. 224, pp. 309-321 (2011).
15. Bugzilla [Online] [Quoted: 20.05.2012] http://www.bugzilla.org/
16. Bičevskis, J.: The Effictiveness of Testing Models. In: Proc. of 3d Intern. Baltic Workshop "Databases and Information Systems", Riga, 1998.
17. Pressman, R.S., Ph.D., Software Engineering, A Practitioner's Approach. 6th edition, 2004.