e-Informatica Software Engineering Journal, Volume 5, Issue 1, 2011, pages: 51–63, DOI 10.2478/v10233-011-0030-4
VERSITA

Experience with instantiating an automated testing process in the context of incremental and evolutionary software development

Janusz Górski*, Michał Witkowicz*
*Departament of Software Engineering, Gdansk University of Technology
jango@pg.gda.pl, miwi@eti.pg.gda.pl

Abstract

The purpose of this article is to present experiences from testing a complex AJAX-based Internet-system which is under development for more than five years. The development process follows incremental and evolutionary lifecycle model and the system is delivered in subsequent releases. Delivering a new release involves both, the new tests (related to the new and/or modified functionalities) and the regression tests (after their possible refactoring). The article positions the testing process within the context of change management and describes the applied testing environment. Details related to documenting the test cases are given. The problem of automation of tests is discussed in more detail and a gradual transition from manual to automated tests is described. Experimental data related to the invested effort and the benefits resulting from tests automation are given. Plans for further development of the described approach are also presented.

1. Introduction

Testing can serve both, verification and validation purposes. It can generate considerable costs (according to [1] it is not unlikely that testing consumes 40% or more of the total development effort) and therefore defining testing objectives and strategy belong to the key decisions to be made in a software development project. This includes not only the answer to how much we want to spend on testing but also what, when and how to test to maximize benefits while keeping the costs in reasonable limits. It is well known that testing cannot prove absolute correctness of a program [2] and can consume (practically) unlimited resources. On the other hand, testing can prove (and does it in practice) that the program includes faults. Therefore, an important criterion to be used in practice is to drive the testing process in a way that increases the likelihood of fault detection. In practical situations it means that we are interested in such test cases selection criteria which result in tests that have the highest potential for detecting significant faults within given time and budgetary constraints. Significant faults are these which have highly negative impact on the program behavior in its target environment. For instance, a fault which is never or very rarely activated within a given operational context of a program is less significant than a fault which is activated in (almost) every usage scenario, unless the former fault is considered to be 'catastrophic' in which case it should be eliminated even for a very high price.

Current tendency is that the number of application programs developed for use in Internet increases rapidly. The complexity of such applications grows. As they are often used to support businesses, commerce and other services, the expectations related to their dependability increase. This requires employment of more sophisticated assurance processes. To maintain their usabil-

ity, such programs have to follow the evolution of their requirements and target environments. Therefore, in addition to the common corrective maintenance practices, they have to be subjected to the perfective and adaptive maintenance processes [3]. A program, instead of being considered as the final result of its development process, is better understood as an object which appears in time in subsequent incarnations, following an evolutionary process. An important question then is not only how to ensure the expected dependability of the program, but in addition how to maintain the assumed level of guarantees throughout the program evolution. Without such change in the attitude we can end up with a product which quickly disintegrates in time and in consequence, the users lose their interest in it.

Changes of a program undermine the confidence in its reliability. Even a very small change (for instance one single bit) can result in a dramatic loss of reliability (for instance, the program stops to work entirely). A widely adopted solution is to have *regression testing* in place, meaning that the program is subjected to a designated set of test cases each time it has been modified. With careful selection of these test cases, a positive result of all tests supports the claim that the reliability of the modified program remains unchanged. Depending on the scope of the changes involved, the regression test suite is being modified to reflect the program evolution.

In present business environments, time is considered a very valuable asset and shortening time-to-market of new products and services is among the highest priorities. This is reflected in the growing popularity of incremental delivery of software products, where the product is delivered as a series of subsequent increments, each increment representing some added value for the users. From the software assurance viewpoint we are facing here the same situation as in software evolution. The product is growing and each subsequent increment is expected not to decrease its dependability comparing to its predecessor.

Combining incremental development with software evolution is the common situation which calls for strengthening software assurance practices and building them into the software process from the very beginning. This has been reflected in the agile approaches to software development where testing is brought to the front of the process by integrating it with the requirements specification (specifying requirements by test cases) and running tests as soon as the first increments are coded.

The purpose of this paper is to present a case study involving instantiation of an automated testing process during development of a substantial Internet application TCT [4, 5] based on AJAX technologies. The application follows the evolutionary and incremental development process model. The data presented in this paper were collected during the period of more than five years of development and evolution of TCT.

The application is based on AJAX technologies [6, 7]. AJAX radically changes the protocol of interaction between the Internet browser and the server. The granularity of exchanged messages drops down from a full page to a page element and such elements are exchanged asynchronously in a way which is highly transparent to the user. The result is that the user has a feeling of working interactively without delays caused by page reloading.

Moving to the AJAX technologies has significant impact on program testing. Numerous techniques, e.g. these described in [8, 9] become non-applicable or can be applied only partially. In Table 1 we assess some of them following [8].

Table 1 demonstrates that only selected techniques (in particular, these based on the so called test recording and replaying) and some tools of the xUnit type (e.g. squishWeb or Selenium) are suitable for testing AJAX-based software. Other techniques are not applicable or require significant modifications.

The TCT application considered in this paper is following the incremental and evolutionary development process. The objective is to deliver subsequent increments while maintaining a satisfactory level of reliability and keeping the development effort in reasonable limits. To achieve this we had to invest in automation of tests which proven to be particularly beneficial.

The paper first sets the scene, explaining the object of testing, its architecture and the pro-

Table 1. Web testing techniques applied to AJAX-based applications [8]

Testing	Adequate	Problems	Tools
Model-based	no	Web models extracted are partial; existing Web crawlers are not able to download site pages	research
Mutation-based	no	Mutant operators are never being applied to client Web code; the application of mutant operators is difficult	not-existing
Code Coverage	partially	It is difficult to cover dynamic events and DOM changes; coverage tools managing a mix of languages are not available	*Javascript*: Coverage validator *Java*: Cobertura, Emma, Clover, etc. *Languages mix*: not available
Session-based	no	It is impossible to reconstruct the state of the Web pages using only log-files	research
Capture/Replay and xUnit	yes	Javascript, asynchronous HTTP requests and DOM analysis are not always supported	*not ok*: Maxq, HTTPUnit, InforMatric, etc. *partially ok*: Badboy, HTMLUnit, etc. *ok*: squishWeb, Selenium, etc.

cess of its development. Then we describe the testing process and explain how it developed in time. Next, we present subsequent steps towards automation of the testing process together with the collected data which characterize the performance of the automated tests. We also highlight the main factors which, in our opinion, had the most significant influence on these results. In conclusion we also present the plans for further improvement of the process as the application grows and its usage context becomes richer.

2. Object of testing

The object under test was an Internet application called TCT, being a part of the Trust-IT framework [4, 5]. The objective of Trust-IT is to provide methodological and tool support for representation, maintenance and assessment of evidence-based arguments. From the user's perspective, TCT implements a set of services. They cover eighteen groups of key system functionalities accessible by a user (listed in Table 6). Multiple instantiations of the services are deployed for different groups of users (presently we run some sixteen such instantiations). Each instantiation supports different 'projects' which are used by different users. The users work independently and in parallel, accessing the services by an Internet browser.

Trust-IT together with the TCT tool has been developed in a series of projects supported by EU 5^{th} and 6^{th} Framework Programmes[1]. The tool was already applied to analyze safety, security and privacy of IT systems and services from different domains, including healthcare, automobile and business. Presently TCT is being used to support processes of achieving and assessing conformance to norms and standards, in particular in the medical and business domains (more information can be found in [10]). Further application domains are being investigated, including monitoring of critical infrastructures.

[1] 5^{th} FR UE Project DRIVE, IST-12040, 6^{th} FR UE Integrated Project PIPS IST-507019, 6^{th} FR UE STREP Project ANGEL IST-033506

2.1. Application architecture

The architecture of TCT follows the *rich client-server model* which is illustrated in Figure 1 [11]. The model includes three layers: database server PostgreSQL [12], application server JBoss [13] and a client written in JavaScript [14] in accordance with AJAX (Asynchronous JavaScript and XML) [6]. The client is automatically uploaded to the browser of the end user.

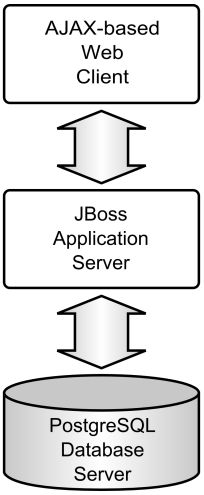

Figure 1. The architecture of TCT

The lowest layer (the database) implements the business logic as a set of stored procedures. The intermediate layer is based on J2EE [15] and links the database with the client. Communication between these layers is based on web services and SOAP (Simple Object Access Protocol) [16].

Each layer is a complex program: presently the database stores some 135 procedures, the intermediate layer and the client layer have 141 classes and 139 classes correspondingly. The two higher layers include additional structure (internal layers) to provide for better understandability and maintainability.

2.2. Increments and evolution

Following earlier prototypes, the development of the present TCT tool was initiated in December 2005. At the beginning, the objectives and the scope of the required functionalities were identified and the delivery of the functionalities was planned as a series of increments. The intention was to follow the *incremental development model* [17, 18] by producing deliverables in subsequent iterations, where each iteration involves requirements gathering and analysis, design, implementation and testing. In effect, each iteration results in the release of an executable subset of the final product, which grows incrementally from iteration to iteration to become the final system.

However, it has been soon realized that following the incremental model in a strict sense is not relevant. As TCT was being developed in the context of on-going research and the research objectives (and consequently the results) were shaped and scoped by the results of the experiments and the feedback received from the participants of these experiments, the requirements for TCT were changing, following a learning curve. Therefore we had to switch to a more complex, *incremental and evolutionary development model*, which can be characterized as follows [19]: it implies that the requirements, plans, estimates, and solutions evolve and are being redefined over the course of the iterations, rather than being fully defined and 'frozen' during the major up-front specification step before the development iterations begin; evolutionary model is consistent with the pattern of unpredictable discovery and change in new product development.

From the testing perspective the incremental and evolutionary development process poses important challenges: (1) to maintain reliability of the subsequent increments it was necessary to have the regression testing in place from the very beginning, and (2) to follow the evolution of the application, the set of regression tests could not be treated as monotonic (i.e. extended by the new test cases reflecting the current increment while preserving the already used test cases); instead it had to evolve following the changes introduced to the already existing functionalities.

So far, TCT has been delivered in eight releases: four of them were related to the *major changes* and the remaining four were related to the *localized changes* of the application. To reflect this scope of change, the former are also called *main releases* and the latter *intermediate releases*. The difference between the two is based on the assessment of the impact of the introduced changes

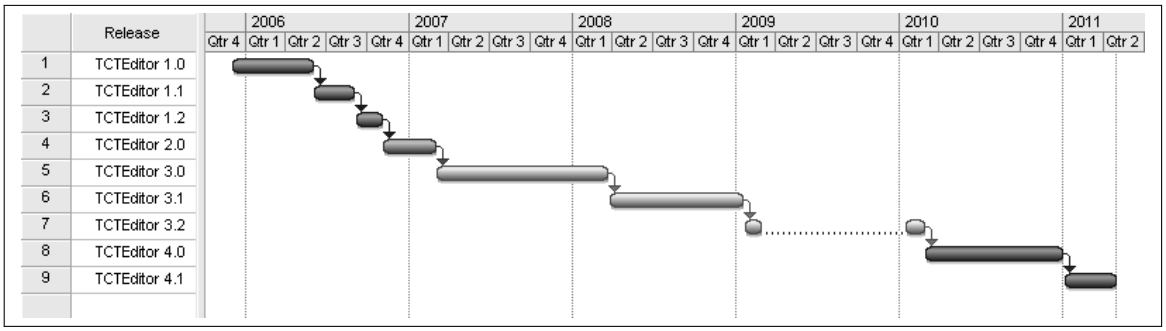

Figure 2. The history of development of TCT

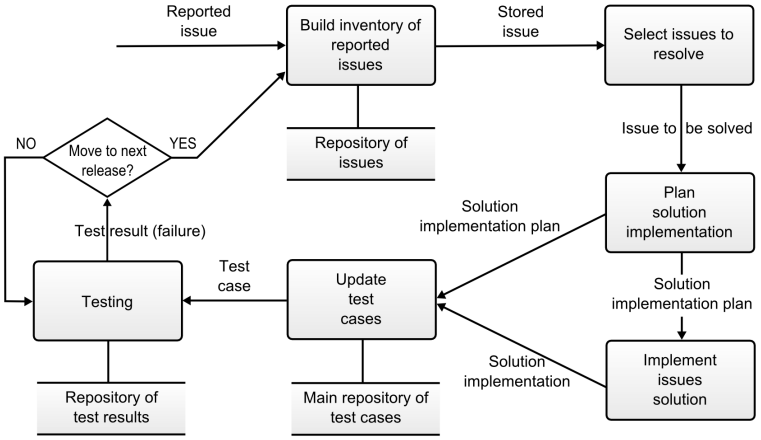

Figure 3. Testing in the context of change management process

and the resulting need for the thoroughness of regression tests. The history of TCT releases is illustrated in Figure 2. The releases are numbered by two digits, where the first one denotes the number of the main release, whereas the second denotes the intermediate release related to the main one.

Throughout the whole evolution, the architecture of the application remains stable. The changes were mainly related to functionality (adding new functions, changing existing functions, removing obsolete functions) and to the applied technologies.

3. Testing in the context of change management

The testing process is embedded in the broader process of change management. The model of the change management process is shown in Figure 3.

The process implements mechanisms for reporting the issues related to the application and for maintaining a related repository of issues. The repository is based on the MantisBT platform [20]. The repository is being periodically reviewed, which results in selecting the issues to be resolved. The selected issues are assigned priorities and then are grouped together in packages, each package containing the issues which can be solved by a common maintenance action. Not all issues require changes in software, for instance some are related to the way a user interacts with the system and can be solved by improving user's documentation and training. The issues which call for software changes are dealt with in the next steps of the process. First, the necessary changes are subjected to planning and then an explicit decision about implementing the plan is bring made, after assessing the resources needed for such implementation and the resulting impact. The plan for the change and the change implementation process provide input to the step of updating the test cases. The new and updated test cases are then stored in

the tests repository forming the new suite of the tests to be performed. The repository of test cases is implemented using Subversion (SVN) [21]. After implementing the change, the tests are being applied.

For a given release of the application, depending on if it is related to a main release or to a intermediate release, the scope of related testing differs. In the former case, the tests include both, the tests covering the new functionality and the full set of regression tests. In the latter case, the tests cover the new/changed functionality and only a subset of the regression tests is included. Limiting the scope of regression tests for a localized change is based on the assumption that the impact of the change is limited and is unlikely to affect the whole scope of the functions.

The testing process is illustrated in Figure 4. It starts with building the *current repository of test cases* which is a subset of the tests maintained in the main repository of test cases. The selection criteria depend on if we are testing a main or an intermediate release of the system. In the former case the current repository is simply a copy of the main repository. In the latter case it is a proper subset of the main repository. Then, the selected test cases are run and the results are collected in the *repository of test results*. The results of the tests are then analyzed and assessed. In case of failed tests, there are two possibilities: (1) inserting new issues to the *repository of issues* (for further processing) or (2) updating the current repository of tests. The former possibility takes place if removing the cause of the test failure involves a significant change; then introducing this change is left to the next releases of the system. The latter possibility is in place if (due to a negative test result) an immediate and localized change is introduced to the software which affects the corresponding tests kept in the current repository. The decision on which alternative is chosen is taken by the tests manager and involves consultation with the representatives of key groups of the users.

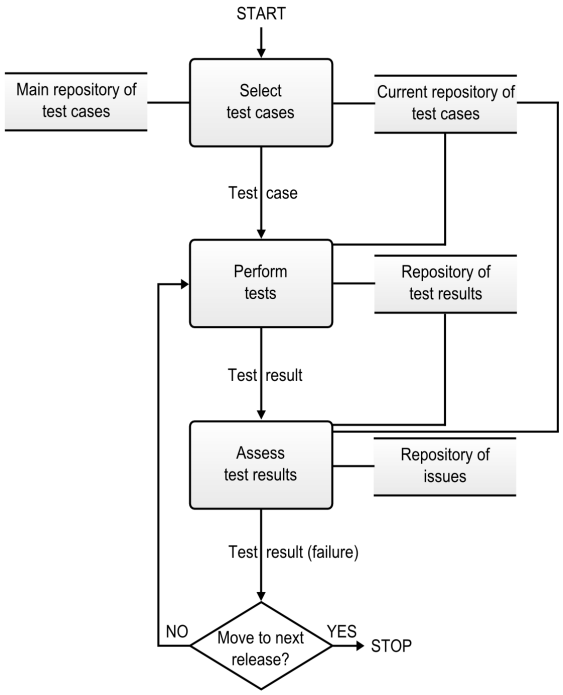

Figure 4. Two phase testing process

After assessing the test results, the decision is being made concerning the continuation of the testing process. If all the issues detected during the previous phase are deposited to the repository of issues, the testing process stops and the next release is delivered to the users. If however, some immediate changes were introduced to the software, the tests kept in the current repository are executed again.

3.1. Specification of test cases

Each test case is represented in accordance with a predefined structure. It includes the following elements:

- **Identifier** - a unique name of the test case. The name suggests the tested functionality;
- **Author** - identification of the person responsible for this test case;
- **Feature** - brief, intuitive description of the tested feature;
- **Scenario** - description of the related testing scenario including:
  - summary of the scenario,
  - description of the initialization phase,

  - the list of actions necessary to complete the test case,
  - description of the closing phase of the test case;
- **Success criterion** - specification of how to assess that the test case completed successfully.

An example test case specification is given in Table 2. The objective of this test case is to check if authentication of the users assigned to different roles works correctly.

3.2. Manual execution of test cases

At the beginning, all test cases were executed manually. The testers were following the test scenarios specified for each test case. If a test case does not pass the success criterion, the tester immediately reports this as an issue to be solved. The issue specification includes details of the sequence of actions which led to the failure.

If the currently reported issue is similar to an issue already reported, then the existing issue description is being updated instead of inserting the new one. Assessment of this 'similarity' was left to the tester. And this appeared to be a weak point in issues reporting. Because analyzing the existing descriptions was boring, the testers often did not go deeply into the details and just concluded that the issue has been already reported and its description did not need any update. The result was that sometimes an important information was not included in the issue description, which adversely impacted fault diagnostics and correction.

4. Automation of test case execution

Manual execution of the process illustrated in Figure 4 consumed significant resources for both, complete regressions tests (full suite of regression tests performed for each main release of the system) and partial regression tests (for the releases related to the localized software changes). This had negative impact on the delivery time of subsequent releases and slowed down the development process.

To deal with the above problems we decided to invest in automation of test case execution. A separate testing system was developed based on the TestNG library [22] and the server and the library offered by Selenium Remote Control [23]. The testing system maintains the TCT system metaphor which is being updated in parallel to the subsequent releases of TCT. The metaphor is used to activate these functions of TCT which are callable from an Internet browser. In consequence, these functions are being activated automatically.

Each result of an automated test case is structured in the XML format (Extensible Markup Language) [24] and inserted to a file. Such reports are periodically reviewed and the detected issues are inserted to the issues repository.

Figure 5 illustrates an example fragment of the code implementing the testing scenario shown in Table 2. Method 1 attempts to log a user in, assuming the role 'viewer'. Method 3 implements the logging sequence. It includes a check if the required element has been visualized. Method 2 finalizes the test scenario and logs the user out.

5. Experiences

So far, there were eight system releases (see Figure 2) including four main releases and four intermediate ones. Testing of the three first releases (two main and one intermediate) was performed by a team of five testers. The process was manual and consumed significant resources. Testing of the second main release was performed by a team of four testers and with partial automation of test cases (automated testing did not play an important role yet and was just experimented with). The third main release was tested with 30% test cases already automated. The testing process involved two testers.

The results achieved were so encouraging that the effort in test case automation was increased which resulted in that for the fourth main release (Release 4.0 in Figure 2) the number of automated test cases reached 95%. This resulted in radical decrease of the effort to execute test

Table 2. An example of test case specification

Identifier	SystemFunctions_Login
Author	Michał Witkowicz
Feature	Access to system functions for different user roles - system login
Scenario	*Summary:* Make sure that all possible roles have accounts in the system (viewer, developer, assessor and admin). Then login to the system as a user assuming different roles. *Initialization:* Check if login screen is properly displayed (find DOM element with id="tct_login_data_form"). If not, the user is likely to be logged in; log the user out (SystemFunctions_Logout). Then, if the login screen is properly displayed - do nothing. *Actions:* 1. Input user login and password. 2. Press "log in" button. 3. Check if the root node named "Projects" of the projects tree is displayed (max. waiting time = 10 sec.). 4. Log out the user. 5. Check if the login screen is correctly displayed (find DOM element with id="tct_login_data_form"; max. waiting time = 10 sec.). 6. Repeat the steps 1-5 for every possible user role: admin, developer, assessor and viewer. *Finalization:* Repeat the Initialization phase again.
Success criterion	Users assigned to all different roles are able to log in to the system

Method 1. test

```
@Test (groups = {"TCT","viewer"})
public void test() throws InterruptedException {
    cmdContainer.loginPage.logIn(viewerUser.getLogin(), viewerUser.getPasswd());
}
```

Method 2. tearDownTest

```
@AfterMethod
public void tearDownTest() throws InterruptedException {
    cmdContainer.mainMenuBar.logout();
}
```

Method 3. logIn

```
public void logIn(String userName, String password)
throws InterruptedException, SeleniumException {
    selenium.type("login_username", userName);
    selenium.type("login_password", password);
    selenium.click("button_logIn");
    cmdContainer.waitForElementPresent("dom=document.
getElementById('projects_root').parentNode.childNodes[3].firstChild",
cmdContainer.loadPageDelay);
}
```

Figure 5. An example test case code

cases and in significant shortening of the delivery time for this release.

The progress in test cases automation is illustrated in Figure 6.

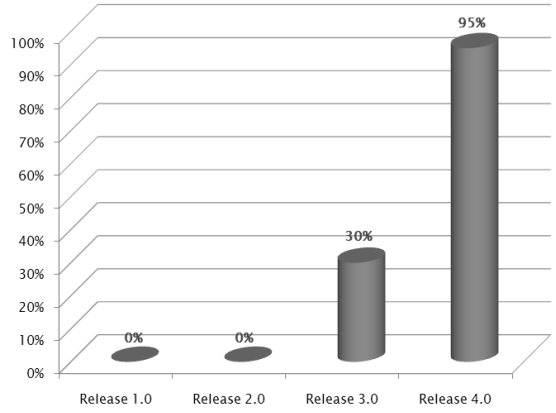

Figure 6. The progress of the test cases automation

Automation of test cases is not free, however. In our experience, one man-day resulted in automation of approximately five test cases (to automate 67 test cases we needed 14 man-days). However, comparing to the effort needed for manual execution of test cases during testing of subsequent system releases, this effort was acceptable (more details are given in Table 4 and 5).

When it comes to the cost of maintaining automated test cases, it depends on a scope of changes in the application in the next release. Obviously, the maintenance of test cases was more expensive while preparing a main release of the system. For example, for the fourth main release it has been observed that one man-day resulted in approximately three updated test cases and total 16 man-days were needed for tests maintenance (see Table 4 and 5). For intermediate releases, usually only few test cases needed to be updated, and the total effort was significantly less.

The total numbers of test cases for the subsequent main releases of the system are given in Figure 7.

Tables 3, 4 and 5 summarize the effort needed for testing the four main releases and illustrate the gain (in terms of effort) resulting from test cases automation.

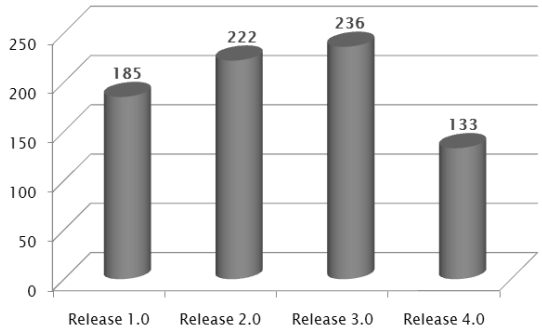

Figure 7. The numbers of all test cases for the main releases of the system

Test cases were following system development and evolution. This was not only because new test cases were being introduced but also because some test cases became obsolete and some other were merged together or modified. As the result, periodic refactoring [25] of test cases was necessary, to maintain test cases integrity and understandability. In particular, such refactoring was performed before testing the release 4.0 which resulted in reducing the number of test cases from 236 (in release 3.0) to 133.

The coverage by test cases of the key system functionalities of the release 4.0 is shown in Table 6.

Tables 7 and 8 illustrate the numbers of issues reported during testing of subsequent main releases (Table 7) and during exploitation of these releases (Table 8).

During the exploitation phase, the reported problems were classified in accordance with the different categories of the maintenance objectives [26]: adaptive, corrective, preventive and perfective. On the other hand, all issues detected during testing were classified as corrective.

Figure 8 compares numbers of different issues detected during exploitation and Figure 9 compares the issues of corrective category between testing and exploitation phases.

From figures 8 and 9 we can see that for release 2.0, testing detected some 30% of corrective issues whereas the remaining 70% were detected in the exploitation phase. For the release 3.0 this proportion looks better and may indicate the positive influence of tests automation. For the release 4.0 this proportion looks much better: automated tests detected nearly 100% of corrective issues. However, it should be noted that the

Table 3. Number of test cases and number of testers involved in testing of main releases

Release #	# of test cases	# of manual test cases	# of automated test cases	# of testers
1.0	185	185	0	5
2.0	222	222	0	4
3.0	236	169	67	2
4.0	133	6	127	2

Table 4. Distribution of testing effort for main releases

Release #	Application design	Specification/ Implementation	Maintenance	Execution	Total effort
1.0	-	5 man-days	0	15 man-days	20 man-days
2.0	-	1 man-day	6 man-days	18 man-days	25 man-days
3.0	10 man-days	15 man-days	5 man-days	20 man-days	50 man-days
4.0	0	14 man-days	16 man-days	3 man-days	33 man-days

Table 5. Distribution of testing effort for automated tests

Release #	Application design	Specification/ Implementation	Maintenance	Execution	Total effort
3.0	10 man-days	14 man-days	0	2 man-days	26 man-days
4.0	0	14 man-days	16 man-days	2 man-days	32 man-days

Table 6. Test case coverage

Functionality	Number of test cases
Copy, cut and paste functions	27
Accessibility of basic system functions	14
Tree of projects and versions	11
Management window for administrators	10
Tree of trust cases	9
Access rights management	9
Appraisal mechanism	8
Management on links	7
Tree element expand and collapse	6
Refresh function	5
Import and export	5
Login and logout	5
Management of user settings	4
Behaviour of tree icons	4
Management of repositories	4
Reference nodes	3
Report generator	1
Traversal tool	1

Table 7. The number of reported issues during testing of main releases

Release #	Issues
1.0	15
2.0	56
3.0	20
4.0	63

Table 8. The number of reported issues during exploitation of main releases

Release #	Adaptive	Corrective	Preventive	Perfective	Sum of issues
1.0	1	47	0	38	86
2.0	4	173	18	75	270
3.0	5	39	1	32	77
4.0	0	1	0	2	3

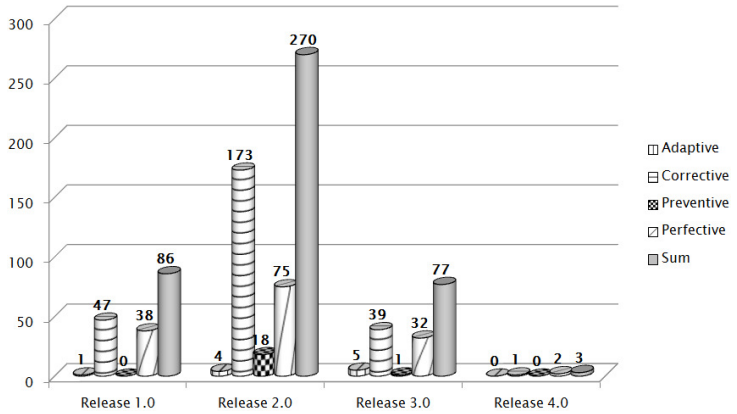

Figure 8. Issues distribution during maintenance of the TCT system

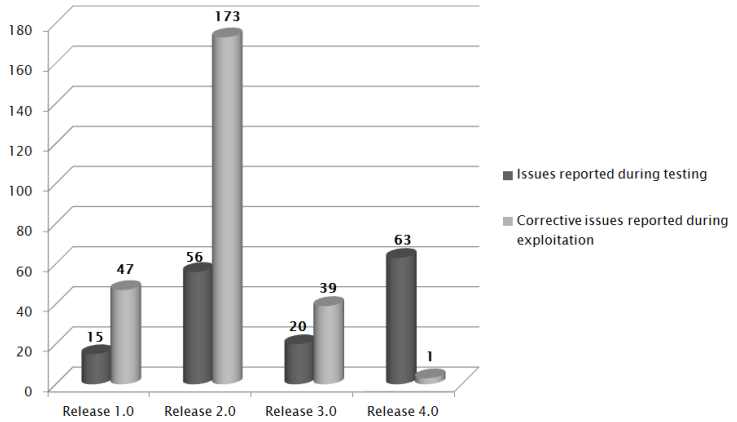

Figure 9. Corrective issues detected during testing and exploitation

exploitation period of release 4.0 amounts for just three months (whereas for release 2.0–12 months and for release 3.0–10 months). To make these data more comparable we calculated the 'issues density' metrics (dividing the number of detected issues by the system exploitation period). The result is shown in Table 9.

The above numbers should not be over interpreted, however. To assess the influence of the testing process on the reliability of the TCT system we would have to take into account other factors for which we have no quantifiable data yet. This involves for instance the influence of the 'size' system change or the operational profile during system exploitation. Nevertheless, at least one relationship can be clearly observed: as the number of different users of the subsequent

Table 9. Corrective issues density for the last three main releases

Release #	Issues per month
2.0	14,41
3.0	3,9
4.0	0,33

system releases increases, in the light of the data presented in Table 9, the claim of increasing reliability of the system gains more credibility.

The presented results suggest that there is still a considerable opportunity to improve the effectiveness of the testing process. However, we cannot ignore the fact that the defects detected during testing are usually the 'big' ones (i.e. such that disable or significantly disturb the usage of the system), whereas the defects detected during exploitation are usually 'small' and rarely prevent the users from using the system. However, it is also worth to note that the difference between 'big' and 'small' defect is context dependent and what is 'small' from one user's perspective (not noticeable at all or slightly disturbing) can be considered 'big' from the perspective of another user with different operational profile. Although we did not yet collected enough data to differentiate between the different impact of the defects, we are well aware of this problem and intend to exploit it while planning for the next steps of test process improvement.

6. Conclusions and plans for the future

The decision about automation of test cases, in particular with respect to the regression tests, has been positively verified in practice as it resulted in considerable reduction of the tests execution effort and contributed to removing subjectivity from execution and interpretation of tests and their results. Despite relatively high cost of the implementation and maintenance of automated tests, the total testing effort decreased and a significant gain in system reliability has been observed.

In our particular case we could observe that perfective maintenance had a considerable share in system changes. This is because the system is being developed in the context of a research process which generates new ideas and discovers new ways of system usage. It can be expected that for systems developed in a business context the influence of this type of changes would be less significant.

In the near future we plan for delivering the next (intermediate) release of the system. This involves the ongoing effort of designing new test cases (checking the new functionalities) and refactoring the existing test cases. Nevertheless, the goal of having 100% regression tests fully automated seems to be not realistic due to the present limitations of the Selenium platform [23].

To exploit the potential of improving the effectiveness of the testing process (illustrated in Figure 9) and to take into account the differences between 'big' and 'small' faults we plan for introducing to our testing process the risk based selection of test cases [27, 28, 29, 30]. This will take into account different usage scenarios and the consequences related to system failure within these scenarios. This information will be then traced back to the system functionalities and reflected in 'weighting' of the related test cases. These weights will be taken into account while planning for the test coverage of critical functions.

The next main release of the system will involve a significant change of technology, especially in relation to the client layer (see Figure 1). To deal with this change we also plan for extending the scope of unit testing of system components. For better control of tests coverage, mutation testing [31] is also considered.

References

[1] I. Sommerville, *Software Engineering*, eighth edition ed. England: Pearson Education, 2007.

[2] R. Patton, *Software Testing*, second edition ed. United States of America: Sams Publishing, 2006.

[3] P. Grubb and A. A. Takang, *Software Maintenance Concepts and Practice*. Singapore: World Scientific Printers, 2003.

[4] J. Górski, "Trust-it - a framework for trust cases," in *Proc. Workshop on Assurance Cases for Security - The Metrics Challenge*. Edinburgh, UK: The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks DSN, 2007, pp. 204–209.

[5] J. Górski *et al.*, "Trust-it research project," information Assurance Group, Gdansk University of Technology (18.10.2011). [Online]. Available: http://iag.pg.gda.pl/iag/?s=research&p=trust_cases

[6] D. Crane, E. Pascarello, and D. James, *Ajax in Action*. Manning Publications Co., 2006.
[7] J. Eichorn, *Understanding AJAX: Using JavaScript to Create Rich Internet Applications*. Prentice Hall, 2006.
[8] A. Marchetto, P. Tonella, and F. Ricca, "Testing techniques applied to ajax web applications," 2007, workshop on Web Quality, Verification and Validation (WQVV), at the International Conference on Web Engineering.
[9] A. Mesbah, "Analysis and testing of ajax-based single-page web applications," Ph.D. dissertation, Delft University of Technology, 2009.
[10] NOR-STA, "Support for achieving and assessing conformance to norms and standards," (04.11.2011). [Online]. Available: http://www.nor-sta.eu/
[11] L. Cyra, J. Miler, M. Witkowicz, and M. Olszewski, "Advanced design solutions of a rich internet application," in *Zwinność i dyscyplina w inżynierii oprogramowania*, A. Jaszkiewicz, B. Walter, and A. Wojciechowski, Eds., Politechnika Poznańska. Poznań: Nakom, 2007, pp. 35–47, (In Polish).
[12] PostgreSQL. (10.06.2010). [Online]. Available: http://www.postgresql.org/
[13] JBoss. (10.06.2010). [Online]. Available: http://www.jboss.org/
[14] D. Flanagan, *JavaScript: The Definitive Guide*. O'Reilly, 2001.
[15] SunMicrosystems, "Developer resources for java technology," (10.06.2010). [Online]. Available: http://java.sun.com/
[16] W3C, "Simple object access protocol," (10.06.2010). [Online]. Available: http://www.w3.org/TR/soap/
[17] C. Larman and V. R. Basili, "Iterative and incremental development: A brief history," *Computer*, vol. Volume 36, no. 6, pp. 47–56, June 2003.
[18] PCMagazine-Encyclopedia, "Iterative development," (06.04.2011). [Online]. Available: http://www.pcmag.com/encyclopedia/
[19] C. Larman, *Agile and Iterative Development: A Manager's Guide*. Addison-Wesley Professional, 2003.
[20] Mantis. (10.06.2010). [Online]. Available: http://www.mantisbt.org/
[21] Subversion. (10.06.2010). [Online]. Available: http://subversion.tigris.org/
[22] Testng. (10.06.2010). [Online]. Available: http://testng.org/
[23] Selenium, "Selenium web application testing system." [Online]. Available: http://seleniumhq.org/projects/remote-control/
[24] W3C, "Extensible markup language," (10.06.2010). [Online]. Available: http://www.w3.org/XML/
[25] A. V. Deursen, L. Moonen, A. Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering*. Sardinia, Italy: XP2001, 2001, pp. 92–95.
[26] E. B. Swanson, "The dimensions of maintenance," in *Proceedings of the 2nd international conference on software engineering*, San Francisco, 1976, pp. 492–497.
[27] R. Black, *Advanced Software Testing - Vol. 2: Guide to the Istqb Advanced Certification as an Advanced Test Manager*. USA: Rock Nook Inc., 2009, vol. 2.
[28] R. Black and N. Atsushi, "Advanced risk based test results reporting: putting residual quality risk measurement in motion," *Software Test & Quality Assurance*, vol. Volume 7, no. issue 8, pp. 28–33, 2010.
[29] R. Black, K. Young, and P. Nash, "A case study in successful risk-based testing at ca," (06.04.2011). [Online]. Available: http://www.softed.com/resources/
[30] F. Redmill, "Theory and practice of risk-based testing," *Software Testing, Verification and Reliability*, vol. Volume 15, pp. 3–20, 2005.
[31] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," CREST Centre, King's College London, Technical Report TR-09-06, 2009.