Effect of Test Completeness and Redundancy Measurement on Post Release Failures – an Industrial Experience Report

Tamás Gergely, Árpád Beszédes, Tibor Gyimóthy, Milán Imre Gyalai
University of Szeged
Department of Software Engineering
Árpád tér 2. H-6720 Szeged, Hungary
{gertom,beszedes,gyimothy}@inf.u-szeged.hu
{gyalai.milan.imre}@stud.u-szeged.hu

***Abstract*—In risk-based testing, compromises are often made to release a system in spite of knowing that it has outstanding defects. In an industrial setting, time and cost are often the "exit criteria" and – unfortunately – not the technical aspects like coverage or defect ratio. In such situations, the stakeholders accept that the remaining defects will be found after release, so sufficient resources are allocated to the "stabilization" phases following the release. It is hard for many organizations to see that such an approach is significantly costlier than trying to locate the defects earlier. We performed an empirical investigation of this for one of our industrial partners (a financial company). In this project, significant perfective maintenance was performed on the large information system. Based on changes made to the system, we carried out procedure level code coverage measurements with code level change impact analysis, and a similarity-based comparison of test cases in order to quantitatively check the completeness and redundancy of the tests performed. In addition, we logged and compared the number of defects found during testing and live operation. The data obtained were surprising for both the developers and the customer as well, leading to a major reorganization of their development, testing, and operation processes. After the reorganization, a significant improvement in these indicators for testing efficiency was observed.**

***Index Terms*—Code coverage, White-box testing, Impact analysis, Test redundancy, Test similarity, Test efficiency, Risk-based testing**

I. Introduction

In risk-based testing, test conditions (features to be tested) are selected based on priority and the likelihood or impact of failure. Decisions on exit criteria (*i.e.* when to stop testing and release the software) are also determined based on risk analysis [1]. Such a risk analysis often includes not only technical aspects such as severity of faults detected, but also non-technical issues like business considerations about when to release the system with new functionality. For example, a bank may decide to release a new version of the software despite being aware of certain defects in the system, in order to reduce the risk of losing customers because they are unable to provide a financial service that competitors of the bank already provide. The drawback of this strategy is, however, that the organization needs to be prepared for an increased number of after-release failures and intensive "stabilization" phases in the software lifecycle.

What is especially interesting is that companies are often unaware of how uneconomical this approach is in the long term [2]. Since the after-release fixes are usually very costly, a modest investment in more efficient testing processes should pay off in the medium term, while not losing the ability to perform a good business risk analysis at the same time. In particular, if appropriate tests with appropriate priorities are performed, this may significantly reduce the risk of after-release failures. This means thoroughly testing the important modified parts of the system (referred to as *completeness* hereafter), and efficiently removing redundant or irrelevant tests (referred to as *redundancy* hereafter).

Although the objective to improve testing and release processes sounds straightforward in theory, it is quite hard to achieve it in practice. It requires in particular code coverage analysis [3] and change impact analysis [4], both areas being technically very difficult in the general case. These methods are usually applied at lower testing levels such as unit testing, but rarely at the system level. However, the above-mentioned risk reduction capabilities can be most effective at higher testing levels, so it is desirable to apply coverage measurement and impact analysis whenever feasible.

In this paper, we describe our experiences on applying the above-mentioned techniques to assess and improve the testing process of one of our industrial partners (a financial organization) through a specific software enhancement project. One novelty of this project was the application of a special *procedure level coverage measurement and change impact analysis*. We defined a set of different coverage measurements based on procedure call and control transfer, which were applicable to large and complex systems and their system level testing. We applied the coverage measures and impact analysis to assess the completeness and redundancy of the tests performed. Similar techniques are usually applied at lower testing levels, and we are not aware of previous publications of a similar combined approach, especially at the system level.

The assessment and improvement involved two phases.

First, we evaluated the efficiency of testing without altering the testing procedure. The outcome of this was a surprise to all concerned, and resulted in a major reorganization of their testing processes. The second phase was carried out four months later. This time we gave suggestions on how to design the test cases based on the uncovered components and redundant tests.

Our key finding was that the efficiency of testing was unacceptably low. Namely, procedure level coverage in the first phase was only 36% with at least 40% of the tests being redundant. However, in the second phase the coverage reached almost 100%. Furthermore, the number of defects reported during and after the tests were logged. These findings also supported an improvement in efficiency; the number of after-release failures halved in the first five weeks.

Overall, these findings showed that even a small enhancement in the process can result in a significant improvement in software quality. Although the company and the technology in question have some quiet special properties, we think that this experience report can be profitable to other researchers and practitioners facing similar problems.

The paper is organized as follows. In the next section we continue with an overview of related work, while in Section III we provide some basic data about the environment and the issues to be addressed. Section IV describes the assessment methodology of the first phase with some technical details, while Section V discusses our methodology for the second phase. Section VI provides a detailed results of our experiments. After, in Section VII we discuss the threats to validity, and in Section VIII we draw some conclusions and describe our future plans.

II. Related work

Code coverage measurement has been used for a long time as a white-box testing technique to evaluate the quality of testing activities. Since the first publication by Miller and Maloney [3], many different studies have been published that elaborate on the relationship between code coverage measures and software reliability [5], fault density or defect coverage [6], [7], and fault detection capability [8]. Weiser *et al.* described the relationship between different kinds of code coverage measures [9], while Malaiya *et al.* showed that there was a clear relationship between code coverage and the efficiency of testing [10].

It is always a problem to eliminate the redundancy of a test, and thus lower the testing costs. There is a significant amount of literature suggesting different approaches to test case selection (or prioritization), which are based on different kinds of measurement. Rothermel *et al.* [11] presented different test case ordering techniques including techniques that use statement and branch coverage information, and they concluded that code coverage can be effectively used for test case selection. White *et al.* [12] applied the firewall testing technique for regression testing to detect late bugs. They saved about 40% of the tests while the testing became more effective; this result is quite similar to ours.

In the present study, we applied procedure level coverage measurement together with impact analysis. It is based on constructing a static call graph [13]. Many researchers agree that building upon a call graph can lead to imprecision in the analysis [14], [15]. However, in many situations it is impractical if not impossible to apply more sophisticated analyses. In this work we extended the traditional coverage variants (branch, decision, statement) to procedure level concepts, which is a novelty. Similar approaches are hard to find. Badri *et al*. also use procedure level dependencies in their so-called control call graph where nodes that do not influence the execution of the procedure calls are left out [16]. The approach based on Static Execute After (SEA) relations also uses a high level program representation, but extends the simple call relationship with other control dependencies, and in this way provides a safer and more complete dependency analysis [17].

III. Environment and goals

A. Overview of the subject organization and the system

We performed this study as part of our research cooperation with a local financial firm which provides various services to their private and corporate customers including financial leasing and real estate loans. The company is one of the market leaders and has many customers, and thus employs a state-of-the-art IT infrastructure to support their operations.

The IT architecture of the company is heterogeneous and consists of various technologies, with a central role of a proprietary technology provided by another local software company. Most of the core systems are built upon this technology, which is an integrated administrative and management system made up of modules (subsystems) using Windows-based user interfaces and MS-SQL database management software. The modules contain programs, and the programs are an aggregate of procedures.

In this study, we focused on a core system based on this technology, referred to as the *subject system* in the following. Specifically, it is a database-oriented application with different layers and a number of external connectors as interfaces to other systems. It has $631,043$ source code lines, 802 programs, $6,201$ procedures and uses 745 SQL tables.

The development and maintenance of the subject system are both performed by the same external software company, which is the owner of the technology. The development projects of this subject system are usually performed in a waterfall-like fashion with semiformal specification and design phases and a very intensive reuse oriented, RAD-like implementation stage. Modest unit testing and a fair amount of integration testing are done by the developer, while most of the testing is devoted to system level testing. The system testing has roughly two main stages. In the first stage (called the 'expert test'), developer and user representatives together test the features of the system. These exploratory tests are based on the technical knowledge of the developer representative and the business expertise of the user representative. Most of the observed failures are fixed in a short time frame without leaving any trace. The second stage (known as 'user test') is the real user acceptance testing which is performed at the site of our partner and by the end users. It is based on a more formal test specification made by business analysts and software architects and validated by the end users. The failures are recorded and – after fixing – are handled with confirmation testing. During user testing new versions of the system under test are deployed on a daily basis.

The decision on releasing the software is usually based on business considerations like deadlines, and the quality assessed by the testing of the software is often a secondary aspect. Still, there are some minimal conditions that have to be met, such as most of the functional tests must be executed, and there may be only a few outstanding critical errors in the system, but the thresholds for "most" and "few" are not precisely defined, for example.

In our case study, the company followed these general practices. They asked us to perform an initial assessment of the processes overviewed above, after which we realized two things that the company also later agreed on: 1) there are several possible ways to improve the processes based on organizational aspects, and 2) there is a significant amount of uncertainty in the testing projects in terms of the completeness and the redundancy of the tests, as no measurement is performed in this respect.

B. The case study and the research goals

Based on the above, we decided to conduct a quantitative assessment of one of the major perfective maintenance projects to find out the actual level of testing quality and provide suggestions for improving the processes.

For this assessment, we chose to apply procedure level code coverage measurements on the selected project and compare these measurement results with the testing activities and with the changes observed in the source code base of the system. For the latter, we employed change impact analysis based on the static code analysis of the modified code partions. Impact analysis is especially important in maintenance testing projects like this (see the previous section on related work). With these measurements, we wanted to address four issues:

1) How complete are the performed tests in terms of the coverage of the changed code?
2) How redundant are the tests in terms of the similarity of the performed tests?
3) Can the testing process be improved using the measurements and observations of the test assessment?
4) How are the coverage values affected when test selection techniques are applied in order to eliminate the redundancy of the tests?

Here the tests were planned based on functional specification only, and no white-box design techniques were applied. Hence out assessment of the tests is in essence a *verification or rejection of the expertise of the test designers based on objective measurement*.[1] White-box techniques are usually applied at lower levels like unit testing, but as this project showed, it is indeed possible to provide useful feedback to the testers about the current efficiency of testing based on coverage and redundancy.

To answer these questions, we performed a two-phase measurement of the testing projects.

a) Phase I.: The aim of the first phase was to assess the testing process of the company. We recorded the execution logs of all of the performed tests and the changes made to the source code during the testing project. The number of recorded defects was also available. We performed four types of coverage measurements, calculated redundancy in other ways and evaluated the bug detection performance of the testing project using the recorded data. In the next section we will provide more technical details on these issues. Through these measurements and by investigating the testing process itself, we were able to answer the first two questions. Both the developer and the customer were quite convinced that the testing was not optimal from both aspects, but they wanted to know the extent objectively. After the evaluation of the results, they both agreed that the processes could be improved in many ways.

b) Phase II.: About four months later, we were involved in another testing project of the same system as part of another major perfective maintenance. Based on the results of *Phase I.*, the testing process was improved to allow the user tests to be proactively controlled based on the results of the measurements. However, this control affected just what is tested (e. g. test cases) and not how (e. g. duration or number of testers). In this phase we applied one selected type of coverage measurement, and did not perform redundancy measurements, but of course we again recorded and evaluated defect numbers. After this phase we were able to answer the third research question too, in a positive and definite way.

In the table below, we provide some basic data about the testing phases of the project.

	first	second
Test duration	3 weeks	4 weeks
Tests performed (executions)	$1,578$	$1,858$
Testers	55 users	60 users
Versions during test	20	23

Table I
BASIC PROJECT DATA

IV. ASSESSMENT METHODOLOGY

In this section, we review the overall approach used to assess the selected testing project and the steps we performed, and

[1]We were able to gather a posteriori information about the redundancy of tests, but this information is rarely available *during* testing. Hence, it was useful to draw the attention of test designers to the problems of the current tests. However, the methods developed are rarely useful in new projects to drive the test design.

provide some technical details about the techniques and the tools we applied.

A. Overview

The first testing project and our assessment consisted of the following main steps:

- **Preparation**
  A new version of the software was deployed for testing purposes. We performed our measurements during the user testing period (see above). The corresponding source code was available as well. **Test plans** were provided by the developer and validated by the user.
- **Daily work**
  Execution logs were collected from all the work stations where the testers performed the tests. **Incident reports** from failing test cases were placed in a common repository. **Source code diff** at the procedure level was calculated using the new version. **Impact analysis** was performed on the source code diff at the procedure level according to the different algorithms described below. **Coverage data** was calculated based on the diff, the impact set, and the execution logs. **Daily changes** were made to the system under test to repair the defects found on the previous day.
- **Final conclusions**
  Completeness and redundancy information was obtained through the statistical analysis of coverage data. Feedback about this information was provided to both the developers and the testers on a regular basis. **Final conclusions** were made after all the test iterations were completed and all the measurement data had been jointly evaluated. This included an investigation of the number of defects identified in the different phases of the life cycle.

B. Data collection for daily measurements

The source code of the changes was available on a daily basis. Changes were detected automatically from the source code repository revisions by creating a textual diff. As the granularity of the measurements was at the procedure level, we identified those procedures that contained any changes made since the last version. The source code was then analyzed, and a static call graph, which was used during impact analysis, was built from it.

Source code programs were compiled to an intermediate representation, and then they were executed by an interpreter. Runtime information was obtained with the help of a modified interpreter, which was able to record entries to and exits from procedures. Calls to built-in procedures were excluded since their source code is not part of the analyzed system. Whenever the interpreter was started a textual log file, which stores the names of the procedures called, was generated. The logs were collected each day during the user test phase.

C. Coverage calculation and impact analysis

As already mentioned in Section II, code coverage can be treated as an indicator of defect coverage in testing. However in maintenance testing projects like this, change impact analysis is essential as well. Both coverage measurement and impact analysis can be implemented at various levels and with varying precision in mind. Due to practical considerations [18], we chose to use procedure level coverage analysis with specifically designed procedure level impact analysis algorithms based on the call graph; however, other coverage metrics and/or impact analysis algorithms would have been sufficed.

Based on the principles of traditional statement level coverage variants and impact analysis [4], [19] we designed practical coverage computation methods. Specifically, in our experiments we applied the following coverage measures:

- **Noimpact**
  This is the simple procedure coverage without impact analysis, *i.e.,* in our measurements *noimpact* coverage is a value that defines what percentage of the modified procedures has been executed at least once during the testing process. It is analogous to statement coverage at the instruction level.
- **Firewall**
  We call the limited impact analysis in which only direct calls are treated *firewall coverage*. The impact set contains the modified procedures and the procedures that directly call or are called by the modified procedures. Then we compute the rate of executed procedures in this limited impact set.
- **Point**
  The coverage value called *point* coverage is based on an extended analysis of the call chains in the call graph. Here, the impact set contains all procedures that directly or indirectly call or can be called from a modified procedure, regardless of the length of the call chain. The coverage value is the percentage of executed procedures within this unlimited impact set.
- **Branch**
  In procedure level *branch* coverage the edges are important, not the nodes. Here we do not count the procedures themselves, but we check the incoming and outgoing call edges in the call graph. The coverage value is the number of the executed call edges of the procedures in the impact set divided by the total number of the call edges of the modified procedures. This kind of coverage measure is derived from instruction level branch coverage.

For an assessment of the completeness of testing, we computed the four types of coverage measures described above using our custom built tool, which takes the source code, the list of changes, and the execution logs as input, and generates the required coverage data.

D. Calculating Redundancy

In order to determine the amount of redundancy in the executed tests we adopted the following principle. We assumed that some of the tests had been executed unnecessarily as other tests checked the program in a similar way; or, in other words, their execution logs were similar (we experimented with different methods to check their degree of similarity).

Of course, the limitation of this approach is that it only takes into account the procedure call graph, which is imprecise due to its inability to analyze statement level control flows, data flows, and so on. Hence, it is not a safe approach, but it is a good approximation.

To examine tests with similar execution properties, we applied *test selection* to get a subset of all executed tests with the property that this reduced set produces about the same amount of coverage as the original, full set. We assumed that if this reduced set of tests had been executed, the same defect detection capabilities would have been achieved, but with lower cost.

We experimented with three different approaches to test selection, which in some cases produced quite different results: two of them attempted to find a subset of all tests as small as possible that represented all the tests, while the third one used a very simple heuristic to reduce a fixed portion of all the tests. The algorithms are described below.

- **Coverage-based test selection**
  In this selection method, the selection criterion was that the reduced set yielded the same noimpact coverage as the full set, while keeping the reduced set as small as possible. Note that there are a number of different possible algorithms available to approximate the optimal solution to this problem. We used a matrix-based algorithm in which the rows and columns represented the modified procedures and the execution logs, and a new row was selected iteratively, and covered columns and irrelevant rows were deleted from the matrix, until the original coverage was obtained.
- **Limited testing effort**
  This was a very simple approach in which we followed the principle that a subset of a fixed size was selected in every case ignoring overall coverage or other similarity measures. However, the subset is not selected randomly, but based on the coverage values. Namely, the top 20% of all tests were selected. In this simple greedy algorithm, the logs were ordered by the average of their four coverage values, and then the top 20% of the logs were selected. The choice for this threshold was rather arbitrary, but here it was the estimated ratio of redundancy values.
- **Clustering-based**
  The principle of this method is to find the classes of very similar tests (based on their execution logs) and to keep just one representative of each class by removing all the others. In this method, we defined a similarity measure between the execution logs, and we applied a clustering algorithm to find the classes. This was the WPGMA algorithm [20].

As the size of the test logs varied by quite a lot, simply getting redundancy values from the number of the reduced tests compared to all the tests is imprecise. Thus, we also computed redundancy in another way: as the total number of procedure calls in the logs of the reduced set divided by the total number of procedure calls in all the test logs (which are basically the sizes of the logs, and which correspond to the lengths of executions).

V. Enhanced testing methodology

After the first testing project we drew several key conclusions, which were then utilized in the second testing project. These are described below.

A. Technical aspects

First, there were a number of technical issues that needed to be addressed regarding the measurement methods. Namely, some issues that possibly harmed the first measurements were eliminated. Source code changes were examined and validated as ‘to be tested’ by the developers. This way we eliminated the problem caused by, for example, dead code which is present in the system but does not affect any business functionality (see more about this in the next section). After handling these issues we found that the coverage measurement was much more reliable.

B. Improvement in test organization

We suggested several improvements to the testing process, which mostly concerned poor documentation, planning and test case design. Most notably, we defined control points in the process, where the measured values were used to affect the other testing cycles. Namely, the coverage values were continuously monitored in a similar way as before, and when the progress (daily growth) of these values decreased for a few days (i. e. the values stabilized), the coverage of each modified procedure was checked. Based on these findings we suggested the creation of new test cases intended to cover yet uncovered procedures. New test cases were then designed and added to the testing process by system architects and test experts.

We did not perform redundancy measurements in the second phase, as it had been decided based on the data got from the first phase that increasing coverage is of prime importance, while redundancy is secondary. Also, in this phase we concentrated on *noimpact* coverage, as our partner asked the testers to attempt 100% coverage for this kind as a start while manually validating the changes — which, given the very low coverage anyway, was a realistic choice at the time.

To evaluate this second project, we compared coverage values and defect numbers with those of the first project. The results of the evaluation are presented below.

VI. Results

Here we present the results of our measurements in four areas. First, we provide measurement data for the coverage values which shed light on the completeness of testing. Then, we give an overview of our measurements on the redundancy, based on the methods described above. Since redundancy measurement is based on skipping irrelevant or less important tests, it can have a detrimental effect on various coverage types. Hence, in this section we also attempt to verify the size of this effect.[2] After, we provide some data about the number of defects found during and after the two testing periods, and the changes in the coverage values between the tests.

A. Completeness

We evaluated the completeness of the first testing project in different ways. In Figure 1, the overall growth tendencies of the four coverage values are presented. It is not surprising that at the beginning the coverage values grow faster, while at the end of testing only a minimal growth of the coverage values can be seen. However, what was surprising for everyone participating in this experiment is how fast this growth decreased (only after about 5–6 days). This means that most of the tests were mere repetitions after about the first 20% of the total time spent on testing (the period was four weeks long).

Furthermore, it can also be seen from this figure that the overall (final) coverage values were rather low (36%, 31%, 27%, and 27% for noimpact, firewall, point, and branch coverages, respectively). What is more, in such a business-critical application as the subject system these values would be unacceptably low! However, the customer decided to release the software as planned because of business risk considerations, and work on an improvement of its processes in the next project.

Nevertheless, we investigated the possible reasons for this extremely low coverage. The first reason we identified was related to so-called dead code. In a large software system like this, changes to business logic imply implementation changes at many different places where the specific logic is implemented, and not just changes associated with specific modules. It can also happen that some parts of the business logic become obsolete, but for some reason their implementation is not removed from the code. However, changes may occur in this 'dead code' too, since the propagation of changes requires it, and the programmers cannot verify if the parts in question are important from a business viewpoint or not. Testers performing user system testing will most probably execute the kind of tests that check those parts of the software that will be often used in real operations, and they will ignore those parts that are used rarely or not at all (dead code). Hence, these parts would not have been important from a coverage point of view, but unfortunately we could not easily eliminate them from the measurement procedure.

The second issue was related to the parallel development branches practised by the developers. Apart from the present testing and release project, they are constantly working on the maintenance of the software version in live operations. So, bugfixes in the live version are ported to all development branches as well. However, it was not the task of the testers of the project in question to test these kinds of modifications.

The final, actually obvious, reason was that the base information for testing was the specification at the business logic level and no white-box information was planned to be involved. Thus, the testers deliberately did not pay attention to checking the list of modifications in the source code of the system.

[2]Note that this is not an assessment of the testing project itself, but of our own test selection approach.

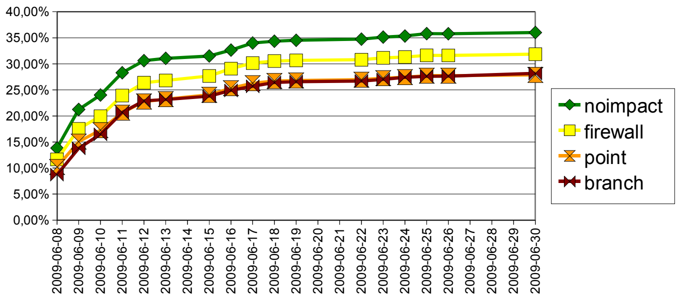

Figure 1. Testing completeness: overall coverage growth

Apart from the overall picture, it was interesting to see the coverage figures on a per-change basis. Changes were made to the system during the testing period, usually on a daily basis. In Table II, we list some statistics about how these daily changes were covered in the case of different coverage measures. We provide the minimal, maximal, and average values (with standard deviation). It can be seen that the values are higher than the overall coverage values given above, but they are still not acceptable.

method	*min.*	*max.*	*avg.*	*dev.*
Noimpact	0%	64%	31.1%	12.3%
Firewall	31%	80%	53.4%	17.5%
Point	12%	64%	38.4%	17.9%
Branch	9%	52%	28.0%	24.0%

Table II
MEASURED COVERAGES OF DAILY CHANGES

B. Redundancy

For the assessment of the redundancy of tests performed, we experimented with removing certain tests that did not significantly contribute to the overall coverage values.

The average redundancy measures obtained by the different selection methods introduced in the preceding section show that the reduction capabilities of the coverage-based selection method (8% of the tests (log files) and 12% of the size (number of procedure calls) are kept) outperform the others on average. This is not surprising since it seeks to select the minimal set of logs that provide the maximum coverage. The reduction was around 90%, which means in effect, that it was ten-fold. The limited and clustering-based methods selected a similar number of tests (20%, 21%) on average, the least effective being the clustering-based method (48% and 59% size). However, we should add this does not mean that coverage-based selection is better than clustering-based selection. On the contrary, as we shall see later on, clustering-based selection has some benefits in terms of its positive effect on other coverage measures.

According to any of the approaches used to measure redundancy, a significant amount (at least 40%) of the test executions could have been saved if a more careful test design had been made and utilized.

In Figure 2, similar redundancy values are shown, but in more detail; for the daily changes and for the procedure call count (log size) variant (the count-based is similar). The values tell us how much of the original tests were selected by our reduction algorithms on each given day.

It can be seen that the redundancies measured on a daily basis are smaller than on a general scale, but they are still significant. We can see in the data that the effectiveness of the coverage-based method greatly varies, but on all but one day it is much better than any other algorithm.

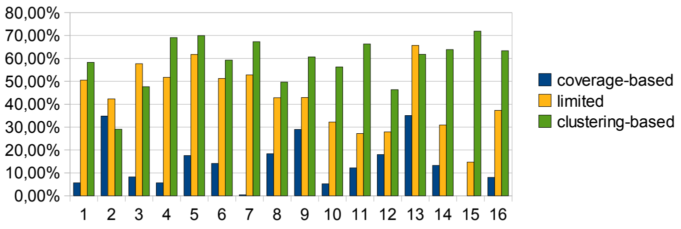

Figure 2. Redundancy values of tests per day (log sizes)

C. Effect of selection methods on coverages

Redundancy measurement is based on different test selection methods. Except for the coverage-based method – whose goal is to achieve the maximum coverage – the selection methods may produce smaller coverages than the ones produced without selection. In the following experiments, we checked these effects in order to get a more complete picture of the different selection algorithms and thus validate our approaches to assess the redundancy (here, we do not assess the test project itself).

These results are summarized in Table III. The measured coverages are the average values of the daily coverage measures expressed in percentage terms. In this table, we compare the original coverage (in parenthesis) with the coverage obtained after being selected by the corresponding algorithms.We provide data for all the different coverage computation methods. The last column for each coverage method shows the amount of coverage percentage "lost" due to selection. In Figure 3, the same values are depicted graphically. It shows the difference between the coverages compared to the unreduced coverage.

	noimpact (31.1%)		*firewall* (53.4%)		*point* (38.4%)		*branch* (28.0%)	
alg.	*abs.*	*rel.*	*abs.*	*rel.*	*abs.*	*rel.*	*abs.*	*rel.*
cov.	31.1	1.00	48.3	0.90	30.9	0.80	20.0	0.71
lim.	27.9	0.90	50.1	0.94	35.4	0.92	25.1	0.90
clust.	26.1	0.84	49.4	0.93	35.8	0.93	25.4	0.91

Table III
COVERAGE VALUES AFTER REDUCTION (AVERAGE OF DAILY VALUES, *abs.* GIVEN IN PERCENTAGE)

It can be seen in Figure 3 that the coverage-based selection algorithm performs better on less strict coverage methods. On the other hand, it is interesting that the clustering-based method produces better results on stricter coverage methods.

Therefore, as a conclusion, although it gives a less effective reduction, the clustering-based approach seems to be most reliable when different coverage-based methods are considered, while – according to its definition – the coverage-based selection is best if noimpact coverage is used.

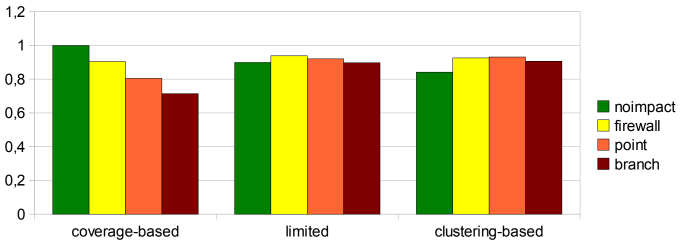

Figure 3. Coverage values after test selection

D. Evaluation of phase II.

The next thing we wanted to do in this experiment was to learn how much improvement in the efficiency of testing and, consequently, defects in the system can be observed by applying code coverage measurement. As we mentioned above, both technical and measurement issues were addressed between the two phases. Thus, in the second phase we used a validated set of changes and an improved testing (measurement) methodology.

The effect of the improvements in the testing methodology, overviewed in Section IV, can clearly be seen in Figure 4. In this graph, the growth of the *noimpact* coverage (the one applied in both phases) can be compared during the two phases. The first phase lasted about three weeks, while the second phase lasted four weeks. As can be seen, the coverage of *Phase I* (lower line) grows for four days, then the growth slows down and even ceases on some days. The overall coverage is about 36% at the end of the testing. The first half of the line of *Phase II* (upper line) looks much like the curve we had in the first project. The decrease in coverage on the 6th and 9th days are due to new features that were added to the system after the testing started, but besides this, the first two weeks show a coverage growth tendency very similar to the first phase. The higher coverage values in this interval are due to the fact that the change sets were validated before inclusion into coverage measurement, which was not done in the first phase. After about two weeks, the detailed coverage values were examined by the developers, and new test cases were designed and included in the testing. It can readily be seen in the figure that after this occurred a steep growth in the coverage was observed, which finally reached 95% at the end, which is significantly better than in the first testing project.

This result itself was convincing enough, but the coverage monitoring and feedback for test design had other, more direct

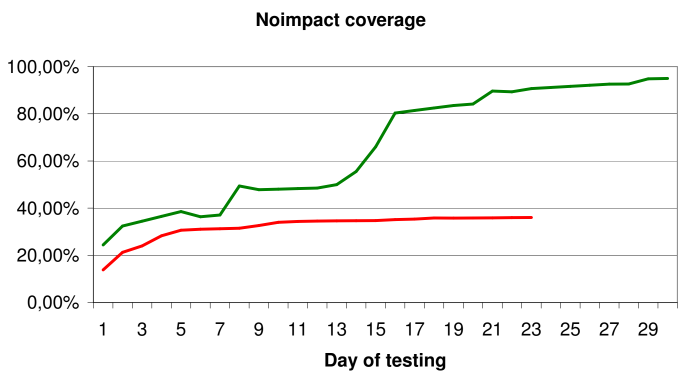

Figure 4. Coverage changes in the two phases (lower line: Phase I, upper line: Phase II)

positive effects as well. Namely, the actual number of defects found in the live system after release decreased. Figure 5 shows the number of reported failures on a weekly basis, for a longer period of time for both testing projects. The two vertical dotted lines show the time of the releases (end of the tests). The continuous line shows the number of defects reported for the live system, while the two short lines before the releases show the defects reported for the test system. Note that due to the nature of test organization (see Section III) it is highly probable that during the so-called expert tests not all defects found were recorded in the incident management system. This may distort the actual number of defects in the test system. As the expert tests are usually performed at the beginning of the testing period, this may be the reason for an unusually low defect rate during the first fortnight.

As can be seen, during *Phase I* the number of defects was quite high in the second week of the test, and dropped sharply low by the end of the testing period. This is in agreement with the findings about coverage growth in the first phase. As for the defects in the live system, a relatively high spike can be observed during the first weeks of live operation. These defects were then only gradualy eliminated in the live system, as predicted by the literature [2].

On the other hand, in *Phase II* the number of new defects during testing built up and remained relatively high until the end of the tests. Then, in the live system another spike was observed in the first week, but it is notably smaller one than in the first phase. After, the defect number decreased fairly quickly to a constant, low level.

Overall, the improvement in software quality in the second project compared to the first one is obvious; the number of after-release defects of the first five weeks dropped from 477 to less than a half (214). Although the causal relationship between high coverage and fewer after-release bugs could not have been proven in this study, the testers at our partner site — based on their previous testing experiments with the same system — are convinced that this relationship exists. All these observations confirmed that by simply monitoring the coverage and taking action when necessary had a beneficial effect on the efficiency of testing and, consequently, on the quality of the software.

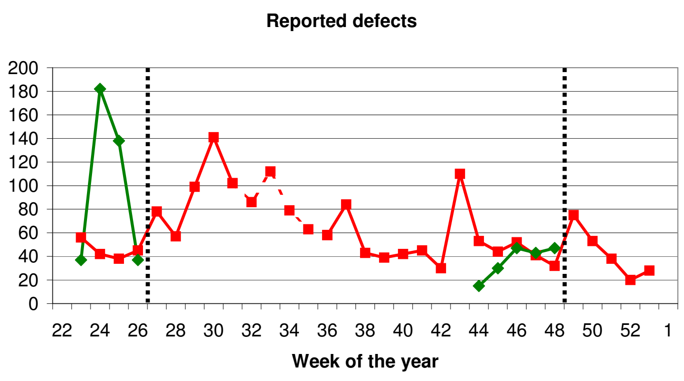

Figure 5. Reported defects in the system

VII. Threats to Validity

We believe that this experience report can be helpful for other researchers and practitioners. It demonstrates that the good practices proposed in the literature are not always easy to implement in practice, and that sometimes the trigger for a change in the processes can be an objective measurement of the current situation, as we did in this project.

However, our conclusions here should not be generalized to other situations too hastily. We identified several possible threats to validity. First, the subject of our case study was a custom-built system based on a proprietary technology which is used within a relatively closed community. Still, since it employs traditional procedural programming and SQL technology, the basic principles and algorithms can be readily generalized to other technologies. Next, although the project was a major development project, the number of measurement data points might still seem modest compared to bigger testing projects. One must not forget either that the redundancy measurements we applied are available only at the end of the testing project, so they are not suitable for driving the test design process (except for optimizing existing regression test suites, for instance).

There are some other technical issues as well. One is that we performed our analyses at the procedure level, and no intra-procedural relationships (like control- and data-flow) were taken into account. This might invalidate our findings about redundancy in some cases, since we compare only the structure of the procedure call relationships and the coverages in the execution logs. Another issue is that we based our measurements on the analysis of different execution logs without taking into account whether these logs were part of different test cases or not. This is because the information about the relationship between the test cases and the actual executions was unavailable (or unreliable). Therefore, we could not draw any conclusions about the efficiency of the test cases themselves, but only about the set of executions. Similarly, no detailed relationship between executions and the defects detected by them were given. Thus, the defect detection ratio of the selection methods were not computed, although it could be the basis of a more precise redundancy value.

Lastly, the number of defects recorded as incidents during the testing phase might not reflect the actual amount of defects found from testing, as already mentioned. This is due to the fact that the testing-fixing-release cycles in this organization allow the testers and developers to fix bugs and continue testing without leaving any trace of these activities. However, this point of uncertainty does not invalidate our findings about defects in the live system (this is the important one!), as these are always faithfully recorded.

VIII. Conclusions and future plans

Here we demonstrated that even in an industrial environment, where business critical applications are developed, tested, and deployed, the efficiency of testing effort is often unknown, and therefore a purely business risk-based approach is adopted when deciding on testing exit criteria. In the long term, however, this may have a significant impact on project costs, since the defects may be found late in the process. However, if a more objective picture was available, the stakeholders would be more motivated to start optimizing their testing processes. The case study presented here shows that the efficiency of such a setting can be rather low, leading to many defects included in (and detected after) the release. By efficiency we mean completeness (how much of the modified code parts were covered) and redundancy (the degree of the similarity among the executed tests) of the tests.

At the beginning of this project, the measured efficiency of the company's testing was very low. In this first phase, the code coverage of modified parts was only about 36% (still less with impact analysis), and at least 40% of the tests were redundantly designed and executed. Moreover, we observed that the efficiency decreased very rapidly only after a few days of the testing project.

All this led to more defects being detected in the live system after the release than in the test system during the testing phase. It was clear to the participants in this experiment that the reason for these results was that there were noticable weaknesses in the process, such as careless test planning including test design and test execution-bugfix-confirmation cycles. Thus, this experiment was evidence to the project owners that the application of such measurements could lead to improvements in the process, so they started to reorganize their processes and involve more white-box techniques and opt for better test planning and control. This is especially interesting given the business criticality of the organization and the subject system, and experienced professional testers.

As part of this reorganization, we helped the company to utilize the data obtained from coverage measurements during test case design, and more actively use this feedback to optimize the testing cycles. To verify this improved testing methodology, we studied a second testing project. We found that the coverage reached almost 100% this time, and that the defect removal efficiency during testing was significantly higher than previously, leading to far fewer failures in the live system (on average less than half of those found previously).

Our partner found these experiments especially useful, and we plan to continue this cooperation by further refining the methods and applying them to similar future projects. However, in order to do this, we should first address some technical questions like developing enhanced redundancy measurement methods that take into account different coverage measurements, and not just *noimpact*. Another research topic is to perform other technology-independent experiments concerning the coverage algorithms we defined earlier in this study. Since we did not find any similar previous studies, we plan to investigate the relationship between these measures and actual defect detection or prediction capabilities and make recommendations about their practical usefulness.

We also plan to continue monitoring our partner's future projects and perform similar measurements to see the overall improvement. Of course, the organization expects economic benefits from the reorganization, so we need to assess the usefulness of the measurement pert and the reorganization from this viewpoint as well.

Acknowledgements

This research was supported by the grants OTKA K-73688, TECH 08-A2/2-2008-0089, and GOP-1.1.2/1-2008-0007. We would like to thank Péter Hlbocsányi and Zoltán Dudás for their help with the experiments.

References

[1] J. Bach, "Good enough quality: Beyond the buzzword," *Computer*, vol. 30, no. 8, pp. 96–98, 1997.

[2] R. Pressman, *Software Engineering: A Practitioner's Approach*. McGraw-Hill Science/Engineering/Math; 6 edition, Apr. 2004.

[3] J. C. Miller and C. J. Maloney, "Systematic mistake analysis of digital computer programs," *Communications of the ACM*, vol. 6, no. 2, pp. 58–63, Feb. 1963.

[4] S. A. Bohner and R. S. Arnold, Eds., *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[5] M. R. Lyu, "Software reliability engineering: A roadmap," in *International Conference on Software Engineering, Workshop on the Future of Software Engineering (FOSE'07)*, May 2007, pp. 153–170.

[6] M. R. Lyu, Z. Huang, S. K. S. Sze, and X. Cai, "An empirical study on testing and fault tolerance for software reliability engineering," in *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE'03)*. IEEE Computer Society, 2003, pp. 119–130.

[7] L. Briand and D. Pfahl, "Using simulation for assessing the real impact of test coverage on defect coverage," in *Proceedings of the 10th International Symposium on Software Reliability Engineering (ISSRE'99)*. IEEE Computer Society, 1999, pp. 148–157.

[8] S. Sampath, E. Gibson, S. Sprenkle, and L. Pollock, "Coverage criteria for testing web applications," Computer and Information Sciences, University of Delaware, Tech. Rep., Apr. 2005, technical Report 2005-017.

[9] M. D. Weiser, J. D. Gannon, and P. R. McMullin, "Comparison of structural test coverage metrics," *IEEE Softw.*, vol. 2, no. 2, pp. 80–85, Mar. 1985.

[10] Y. K. Malaiya, N. Li, J. Bieman, R. Karcich, and B. Skibbe, "The relationship between test coverage and reliability," in *Int. Symp. Software Reliability Engineering*, Nov. 1994, pp. 186–195.

[11] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Trans. Softw. Eng.*, vol. 27, no. 10, pp. 929–948, 2001.

[12] L. White and B. Robinson, “Industrial real-time regression testing and analysis using firewalls,” in *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004)*. Los Alamitos, CA, USA: IEEE Computer Society, Sep. 11–14, 2004, pp. 18–27.

[13] B. G. Ryder, “Constructing the Call Graph of a Program,” *IEEE Trans. Softw. Eng.*, vol. SE-5, no. 3, pp. 216–226, May 1979.

[14] J. Law and G. Rothermel, “Whole program path-based dynamic impact analysis,” in *Proceedings of the 25th International Conference on Software Engineering (ICSE’03)*, May 2003, pp. 308–318.

[15] X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley, “Chianti: A tool for change impact analysis of Java programs,” in *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA’04)*, Oct. 2004, pp. 432–448.

[16] L. Badri, M. Badri, and D. St-Yves, “Supporting predictive change impact analysis: A control call graph based technique,” in *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC’05)*. IEEE Computer Society, 2005, pp. 167–175.

[17] J. Jász, Á. Beszédes, T. Gyimóthy, and V. Rajlich, “Static execute after/before as a replacement of traditional software dependencies,” in *Proceedings of the 2008 IEEE International Conference on Software Maintenance (ICSM’08)*. IEEE Computer Society, Oct. 2008, pp. 137–146.

[18] S. Elbaum, A. Malishevsky, and G. Rothermel, “Test case prioritization: A family of empirical studies,” *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 159–182, 2002.

[19] L. White and K. Abdullah, “A firewall approach for the regression testing of object-oriented software,” in *10th International Software Quality Week (QW’97)*, May 1997, p. 27.

[20] A. K. Jain and R. C. Dubes, *Algorithms for Clustering Data*. Prentice Hall College Div, Mar. 1988.