Enhancing Software Testing by Judicious Use of Code Coverage Information

Stefan Berner
Swiss National Bank
Zürich, Switzerland
stefan.berner@snb.ch

Roland Weber, Rudolf K. Keller*
Zühlke Engineering AG
Zürich-Schlieren, Switzerland
{row, ruk}@zuehlke.com

Abstract

Recently, tools for the analysis and visualization of code coverage have become widely available. At first glance, their value in assessing and improving the quality of automated test suites seems to be obvious. Yet, experimental studies as well as experience from projects in industry indicate that their use is not without pitfalls.

We found these tools in a number of recent projects quite beneficial. Therefore, we set out to gather code coverage information from one of these projects. In this experience report, first the system under scrutiny as well as our methodology is described. Then, four major questions concerning the impact and benefits of using these tools are discussed. Furthermore, a list of ten lessons learned is derived. The list may help developers judiciously use code coverage tools, in order to reap a maximum of benefits.

1. Introduction

Code coverage analysis and visualization tools – e.g. [2][4][5][9][12] – have recently become an affordable and integrated part of various development environments. These tools typically allow for both, the measurement of various aspects of code coverage as well as the analysis and visualization of those parts of the code that have (or have not) been touched after a specific piece of testing code has been executed. Quite often, these tools are used to assess and improve the quality of an automated test suite. At first glance, the value of code coverage measurement, analysis, and visualization seems to be obvious. On second thought, their use is not without pitfalls. The experiments of Hutchins et al. [3] suggest that a very high level of coverage is necessary for the correlation 'higher coverage implies higher defect detection' to become evident. Yet in real world projects, coverage levels beyond 90% are often difficult or simply too expensive to achieve. On the other hand, the coverage levels required by an organization's quality assurance team are often too low to have any significant effect. Furthermore, Lawrance et al. [7] report that developers using these tools tend to become overconfident in their testing efforts, thus neutralizing or even negating their potential benefits. According to the authors' experience, these pitfalls can be observed in many projects.

However, in a number of recent projects we found code coverage analysis and visualization tools quite beneficial. Provided that they are properly introduced and judiciously used, we claim that they can contribute to a more efficient development of automated tests.

In this report, we aim to substantiate this claim with data from one of our projects, which served as a sort of catalyst for this work[1]. In the project, automated (unit) testing had been well established from the outset, and a code coverage analysis and visualization tool was introduced in the last quarter of the construction phase[2]. This setup allowed for a close (post mortem) examination of the impact that code coverage analysis and visualization had exerted on the development process and especially on automated tests.

Specifically, the following four questions related to the introduction of code coverage analysis and visualization tools will be addressed:

(1) What was the impact in terms of code coverage of the automated tests?
(2) In comparison to the pre-tool-state, which group of developers – experts, seniors, juniors, etc. – benefited to which degree from coverage analysis and visualization?
(3) What were the benefits from a qualitative point of view?

---

1 The project was carried out when the first author was affiliated with Zühlke Engineering AG.

2 The project was organized according to the Rational Unified Process into the four phases: Inception, Elaboration, Construction, and Transition.

* The third author carried out part of this work as Adjunct Professor at Université de Montréal, Canada.

(4) Did the use of coverage visualization increase the amount of modifications in the source code?

Note that this paper is an experience report and not a research paper. The data at hand was gathered from one single project and not from a series of projects. It is not the result of a controlled experiment, but was produced in a kind of post mortem project analysis. Hence, there is no control group. We took great care, though, in extracting and processing the data, yet cannot guarantee the complete absence of smear or noise effects. However, because the data stems from a real world project, and the purpose of this report was not to prove or falsify a certain hypothesis, the setup is scarcely biased by the nature of the hypothesis and the expected results.

Below, we first outline the setup of the project under examination. Then, the tool introduction and data gathering is described. Thereafter, in Sections 4 through 7, the four questions stated above are discussed. Furthermore, in Section 8, a list of ten lessons learned is presented. Section 9 concludes the report and provides an outlook into future work.

2. Project Setup

2.1. Purpose of the System

The project examined in this report was part of a service outsourcing initiative, where a bank outsourced its securities trading to another bank specialized in this kind of trading. The purpose of the system was the integration of the service provider's securities trading system with the bank's backend systems responsible for processes like accounting, output management, reporting, archiving, etc.

Besides the 'usual' risks involved in this kind of projects (for example, numerous and often fragile system interfaces), two additional risks required special attention. First, the service provider implemented a new securities trading system in parallel. Second, the

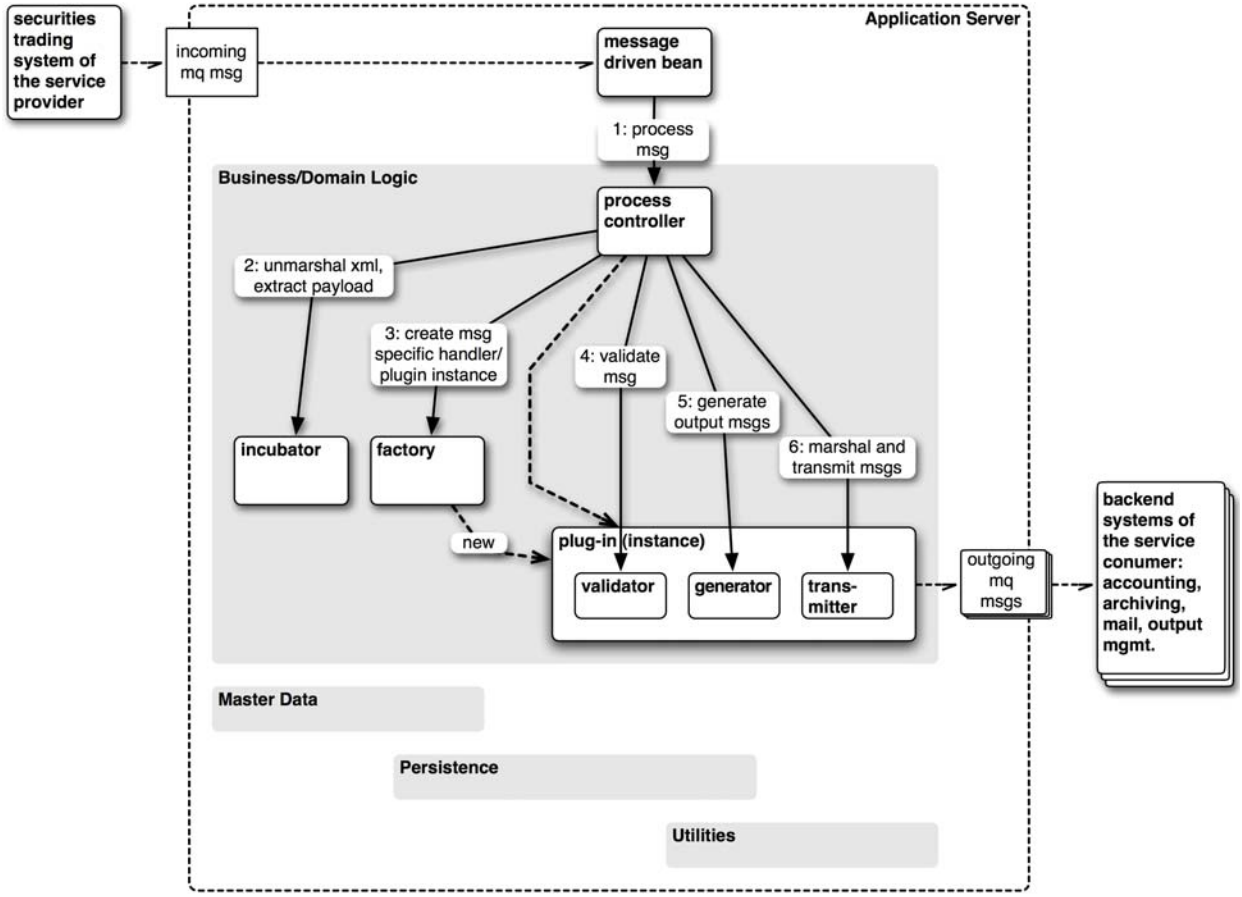

Figure 1. Architectural overview. Business logic is layered atop components to handle master data and persistency. All components use common utility functionality. For the business logic components, some detail is shown in form of an informal object collaboration diagram.

integration of the backend system was the ry) pilot for the messaging infrastructure, a newly troduced enterprise service bus.

Briefly speaking, we had to build a highly adaptable system that processes trading and housekeeping messages (orders, trades, counter trades, cancellations, corporate actions, etc.) from the service provider's securities trading system, and generates and transmits data (accounting records, vouchers, etc.) for the appropriate backend systems, depending on message type and content. Altogether there were about 250 different types of incoming messages as well as about 10 backend systems and system interfaces to integrate.

2.2. System Architecture

The system was designed as a hub-and-spoke architecture [10] with a core to control the common processing and a configurable mechanism to integrate plug-ins in order to handle message-specific parts. The plug-ins were intended to separate concerns and isolate changes in the input data. As already mentioned, the service provider's trading system was subject to an ongoing implementation. Our process and product had to cope with this situation.

The message processing was divided into four steps – incubation, validation, generation, and transmission. A plug-in was responsible for the handling of the message specific parts of each of these processing steps. Each plug-in could be assembled from predefined/reusable or newly developed parts.

The core was responsible for controlling and orchestrating the processing, transaction management, writing logs and audit trails, making data persistent, as well as diagnosis and housekeeping, e.g., end-of-day processing (see Figure 1).

2.3. Test and Testware Architecture

By intention there was no dedicated testing phase, as testing was considered an integral part of the usual development activities. The test strategy focused on two aspects: automation and early end-to-end testing. Hence, the classical approach towards automated unit testing [6] was supplemented with tests of the complete system (according to its actual degree of completion).

Much emphasis and effort was put into the testware to allow individual developers to do efficient end-to-end testing. This included:

- A set of sophisticated mocks to simulate the messaging infrastructure of the underlying enterprise service bus. This allowed for an immediate end-to-end test by the developer without the need to deploy the system after each change.
- A user interface [2] to run all automated tests independently from the development environment. This allowed (a) for conveniently running the same tests after each deployment to the pre-production environment and (b) for the project manager and analyst to keep track of the system's current status and capabilities. This approach was quickly adopted by another, independent team within the bank.
- A configurable comparator to compare the generated (xml) output messages with a reference message. The configuration of the comparator allowed for a quick and straightforward specification of those message parts that ought to be ignored when an actual result is compared to the expected result. In this way, a message's ever-changing sequence number, for instance, could be ignored.

It was planned to use a coverage analysis and visualization tool right from the beginning, but the tool included in the development environment did not work in our context. Yet, a working coverage analysis tool [4] became operable in the last quarter of the construction phase.

2.4. Team Structure and Task Allocation

The core team consisted of eight persons: project manager, business analyst, architect and chief programmer (the first author of this report), three senior developers and two junior developers. It was complemented by experts for the enterprise bus/asynchronous middleware, who could be called in on demand.

The riskiest parts were clearly the business/domain logic, which had to be fast, stable and adaptable. Especially the message types and -format coming from the trading system were expected to be unstable during a considerable part of the project. To cope with this problem, the most experienced resources where assigned to the business logic component.

The junior developers – both of them senior host developers recently transferred to the Java development department – were assigned to the components responsible for the handling of master data and persistence. These components were considered to be of lesser risk, as requirements were stable and the interfaces internal to the system.

It should be noted that the assignment was not strict, but merely a focal point. In principle, the code was owned collectively and each team member (and

his or her tasks) did have one other team member who could serve as a back up.

3. Tool Introduction and Data Gathering

Lawrance et al. state that the usage of code coverage tools (can) cause developers to overestimate their test effectiveness [7]. In previous projects, we made similar experiences, especially when the usage of such tools was not introduced properly. Bare provisioning of the coverage analysis and visualization features often leads developers to compete over the highest possible coverage rates without using the visualization to reflect about more effective tests. The coverage rate may increase, but the tests are not improved in quality. In order to avoid this pitfall, the team was coached with the tool, heuristics were given to identify areas that remain often untested, such as error handling (see [1]), and it was made clear to the team that the goal was not higher coverage at all costs. It should also be noted that the coverage analysis tool was introduced into a project with an operable test suite of around 240 automated tests.

Gathering of coverage data was done through the coverage analysis tool. To get coverage numbers of the pre-tool era, we restored previous baselines/revisions from the repository (in our setting: cvs) and simply replayed the complete test suite to reproduce the result of the testing done to this specific baseline/revision. This was possible because source code, test code, test cases, test data, and expected test results were all put together in the repository and tagged with baseline/revision information.

Data gathering of the change rates was done through a cvs history dump and a spreadsheet. For any addition, modification or deletion of a repository file, the history dump contained the date/time and the responsible developer. This data was imported into the spreadsheet for post-processing, e.g., aggregating/counting all changes within one week.

4. Impact on Code Coverage

Figure 2 shows various coverage measurements for the whole system over a period of half a year – three months before the coverage analysis tool was introduced and three months after. The coverage analysis and visualization tool was introduced in mid-January 2006. From that time on a clear increase in statement -, branch -, and method coverage is visible. During the period observed, all three (statement, branch and method) coverage measurements always move up or down in concert. We therefore will not differentiate

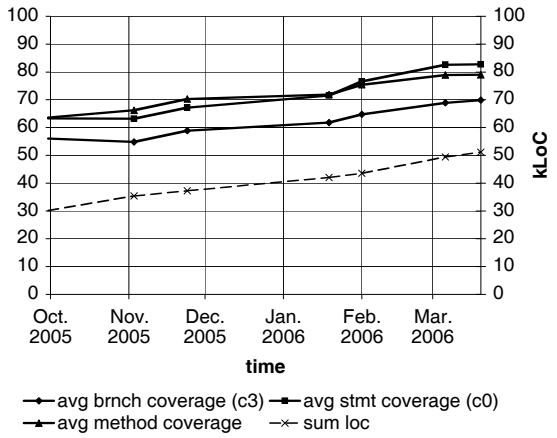

Figure 2. Branch - (c3), statement - (c0), method coverage, and lines of code (loc) for the system.

them in the context of this report and simply talk of code coverage.

Before coverage visualization became available to the development team, the coverage rate increased very slowly up to around 70%. We often observed this behavior in systems where the architecture makes it difficult to do comprehensive testing beyond a certain level. This typically occurs when not all the conditions necessary to enter a specific piece of code can easily (and automatically) be replicated through tests.

As change requests and new test cases impacting the changed parts could be implemented quite easily, we assume that the architecture of our system was not the true limitation. Nevertheless, the coverage rate stalled at around the level of 70%. We attribute this to the developers, who were sufficiently confident about the quality of their tests. They could not think of anything else to test, as long as they did not dispose of any effective aid for detecting aspects not covered yet.

In terms of absolute numbers of test cases, there were around 240 before the introduction of the tool. The number of test cases increased quite quickly to 317, and then to 420. However, this change cannot be completely credited to the exploitation of coverage visualization and to the new possibilities for scrutinizing the tests and the productive code. In fact, new plug-ins to handle new message types were developed and tested in parallel. Thus, we estimate that roughly one third of the additional test cases are due to the implementation of those plug-ins, and that the other two thirds can be attributed to coverage visualization.

5. Impact on Developers

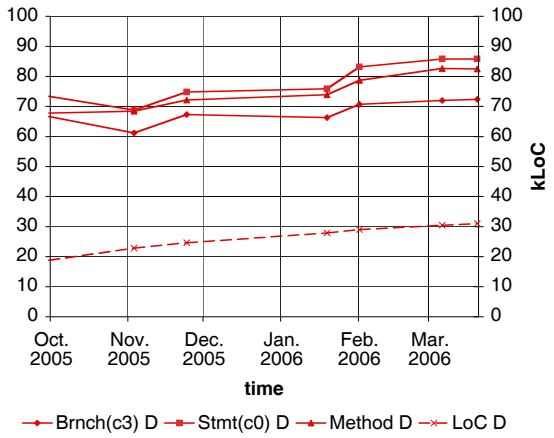

Figure 3. Statement - (c0), branch - (c3), method coverage, and lines of code (loc) for business/domain logic components developed by senior developers.

As mentioned in Section 2.4, the most experienced developers were assigned to the business/domain logic components, whereas the more junior ones were assigned to master data and persistence components.

In retrospection and by pure chance, this setup allows for an analysis of the degree to which developers with different experience levels benefit from code coverage visualization. The correctness of the analysis is based on the assumption that the testability (and thus the effort to create tests) of the different components is similar. Interviews with the project team – which includes the first author – support this assumption.

5.1. Senior Developers

For the senior developers the coverage rate stalled before the tool was introduced. After the introduction, the coverage rate increased immediately, but only by a moderate amount. Roughly a month later, the coverage rate slowed down again (see Figure 3).

According to the developers' comments, we concluded that coverage visualization helped them to identify tests that had been missing before (according to their individual judgments). Usually, this meant new or extended tests for error handling, synchronization and robustness. It rarely meant new or improved tests of special conditions of the blue-sky behavior. After they internalized this knowledge, it became part of their mindset to write tests. They still used (and enjoyed) coverage visualization, but in essence, they had no further need for the tool. Coverage visualization revealed subsequently fewer new aspects they found 'test-worthy'. Rarely used and automatically generated setter or getter methods were for example not considered to be 'test-worthy'. As mentioned before, the goal of coverage analysis and visualization was not to reach an ultimately high coverage rate, but to improve the tests by including missing yet important aspects.

5.2. Junior Developers

For the junior developers we made two distinct observations. In the first place (see Figure 4), similarly to the senior developers, code coverage visualization leads to a visible increase of the coverage rate. They needed around two weeks longer to become comfortable with the coverage visualizations, but in comparison to the senior developers, the duration of the effect was longer and the relative increase of the coverage rate was a lot higher. After a short time, the junior developers reached the same coverage rate as the senior developers. In contrast to the senior developers, not only tests for robustness and error handling were added, but also a large number of tests for checking special cases in the blue-sky behavior. A closer inspection reveals that an experienced developer often adds such special cases and their tests more or less automatically, in order to make the system behave according to the 'principle of least astonishment' [1][14]. Another factor contributing to the steep increase of the coverage rate was due to a couple of test cases that were not integrated in the embracing test suite. The coverage measurement led the developers to revisit these test cases and either integrate them in the embracing suite or to delete them altogether.

As a second observation (see Figure 5), the coverage rate increases for a very short time and then drops significantly. Missing or inconsistent error handling always plagued the corresponding component a little

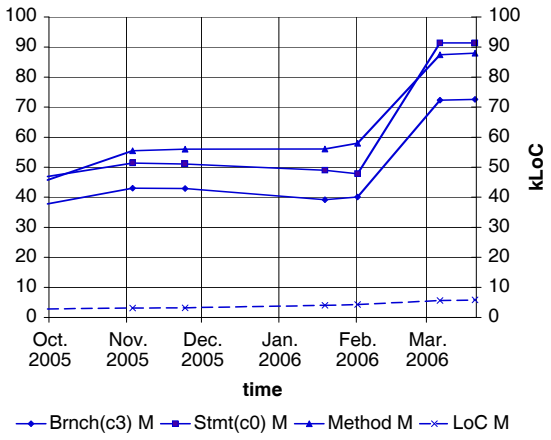

Figure 4. Statement - (c0), branch - (c3), method coverage, and lines of code (loc) for master data components developed by a junior developer.

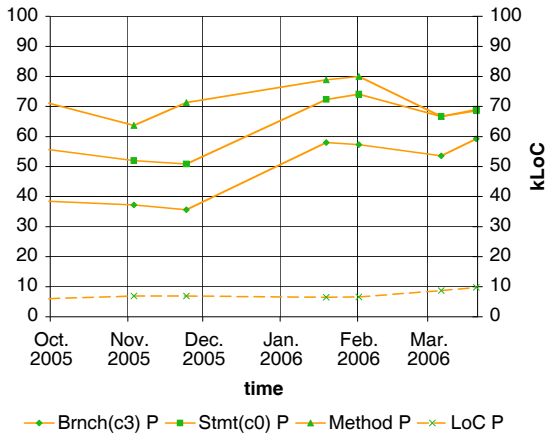

Figure 5. Statement (c0), branch - (c3), method coverage, and lines of code (loc) for the persistence logic components developed by a junior developer.

bit. The coverage visualization caused the developer (at least he told so) to rethink the internal design. He decided for radical changes, refactored the structure massively and introduced a persistence framework. As a consequence, the coverage rate dropped during the rework for a certain time, before it rose again as the component stabilized.

5.3. Collectively owned Component

Besides the components with a clear assignment to a certain developer resource, there was one component – the utility component – that was owned and developed collectively. This component provides common business-independent functionality like logging and helpers for exception handling, as well as a couple of utilities, e.g., string manipulation utilities.

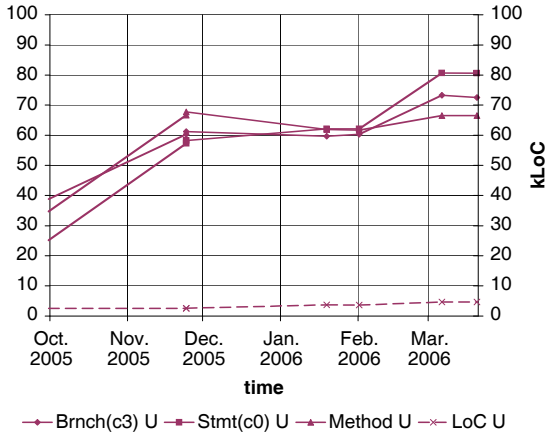

Figure 6. Statement - (c0), branch - (c3), method coverage, and lines of code (loc) for the collectively owned component containing common utility functionality.

Naturally, these mixed-bag-components and their tend to be neglected by the time, and indeed, the coverage rate was slowly decreasing (see Figure 6). In contrast to other components where almost no intervention by the architect or project manager was necessary to motivate developers to improve their automated tests, this component was left aside, at least at the beginning (not visible in the figure because of a missing measurement point in mid-February). Thereafter a mix of the effects described above occurred). The coverage rate increased, but the increase is right between the increase of the senior and junior developers, which supports the analysis.

6. Benefits of Code Coverage Visualization

Based on experiences from this project and previous ones, we identified three prominent areas of benefit from code coverage visualization:

- Improvements of code robustness
- Consolidation of the automated test cases
- Detection of new defects, mainly in the error-handling

6.1. Improvements of Code Robustness

Not everything that is detected through code coverage visualization manifests itself as a defect. The robustness and understandability of the internal contracts and communication of a component often facilitates or prevents the introduction of defects. A component or class that refuses to behave according to ‘the principle of least astonishment’ and/or with a fragile error handling may be correct when examined in isolation, but in cooperation with other components, it may well be a steady source of defects. Tests limited to the blue-sky behavior as well as the lack of tests of the handling of special cases and errors both support the perpetuity of such problems. The testing in an integrated, end-to-end context (instead of unit testing of single classes) together with code coverage analysis contributed substantially to identify these sub-components.

Often a large number of untouched conditions and disproportionateness and imbalance between statement and branch coverage rates are signs of such components. In our experience, this is an area where especially the junior developers took great benefit from the coverage analysis and visualization (see Section 5.2).

6.2. Consolidation of Automated Tests

As the automated test suite grows, it is often inevitable that the same aspect is tested by several distinct test cases, e.g., the error handling for a message with a corrupt message format is always the same, independently of the message type. For the developer this coherence is not always obvious, and as a consequence, some test cases are dispensable. During test execution, these dispensable test cases do no harm, and there is no need to care about them. However, when the system has to be changed or refactored, e.g., due to changing requirements, not only the productive code but also the test cases are subject to change. Contrary to the productive code, the automated test suite is often not designed to cope with those changes.

Coverage analysis and visualization helped developers in two ways to recognize dispensable test cases. First, the visualization showed that eligible code may already be covered. Second, the tool we used also showed how often each line was executed. As a consequence, developers are able to reason about and decide (a) whether the existing coverage is sufficient or (b) if it is still necessary to add further test cases, or (c) to initiate a refactoring of the test suite to make it more robust against changes. All three options were performed and led to a sleek but effective test suite.

Another effect was the assessment of 'loose' test cases. These test cases, being not part of any larger unit of test cases, were either integrated into an embracing suite or deleted altogether.

6.3. Detection of New Defects

Before the introduction of coverage analysis and visualization, the existing tests covered a lot of the blue-sky behavior. The developers preferred to create end-to-end tests using the given testware (see Section 2.3) instead of writing fine granular tests of individual classes. As a consequence, the error handling (including error messages) was quite good at this end-to-end level, and so was the public interface of the business/domain logic component, as this interface was close to the system border.

However, inside the system the error handling between the business logic, master data, persistence and utility components was inconsistent. In this context, a number of defects where detected which (a) would have stopped the system without need, (b) did not stop the system despite the need to do so, or (c) reported inconclusive error messages. Furthermore, defects in the handling of special cases especially in the persistence component were detected. The continuing pressure to handle these cases properly urged one of the junior developers to refactor his component (see Section 5.2).

Typically, many defects detected using coverage visualization were related to error handling and to spe-

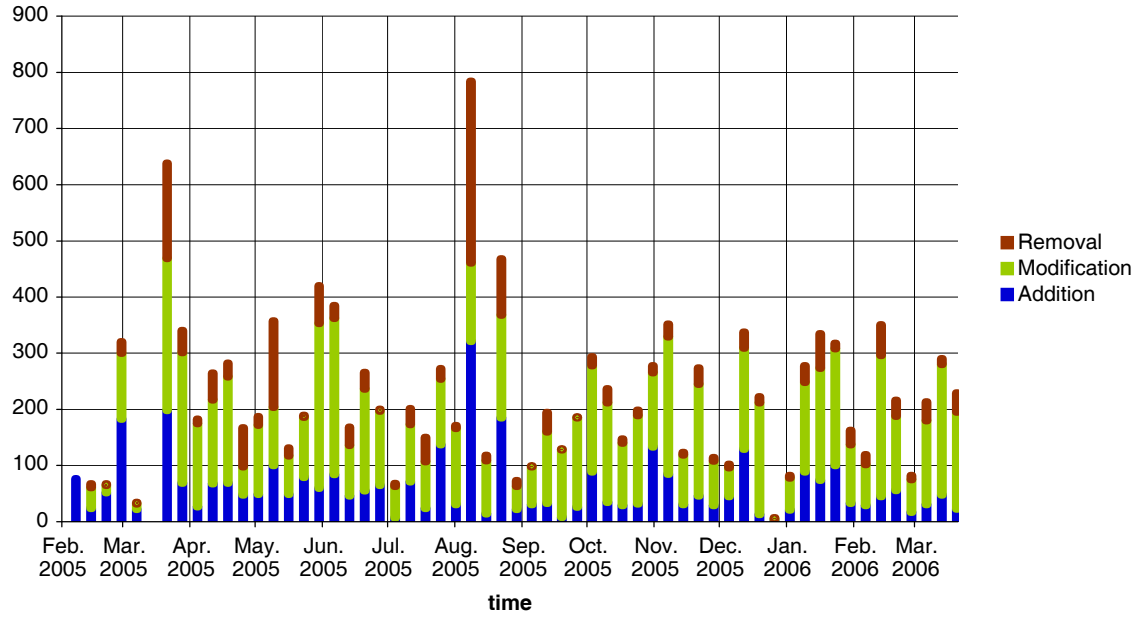

Figure 7. Accumulated number of weekly removals, modifications and additions of files in the repository during the first author's involvement in the project (January 2005 until March 2006).

cial cases in the blue-sky behavior. A couple of defects related to non-functional requirements were detected as well. Defects concerning the typical (blue-sky) behavior were rarely detected.

An increased defect rate was clearly observable in the test management tool. Yet, we were not able to derive a clear and quantifiable relationship to coverage analysis and visualization, since the defect rate was also influenced by (a) a more formal handling of defects in the later phases of the project where increasingly more defects were tracked, (b) the early use of the system in a pre-production environment, and (c) the continuing development of plug-ins by the maintenance team.

7. Impact on Code Change

Except for a couple of days to introduce the code coverage analysis and visualization tool, the original schedule of the project plan was not affected in any way by the use of the tool. Above, in Sections 4 and 5, data about the impact on code coverage rates was shown. In this section, we will present some data about the impact in terms of changes to the system.

Figure 7 shows the accumulated weekly change rate over the time the authors accompanied the project. The big peaks in March and August 2005 indicate major refactorings of the core controlling the processing and accessing the plug-ins (cf. Sections 2 and 3). The "calm" areas in between indicate phases where mainly message specific plug-ins were developed. Surprisingly, the change rate did not alter noticeably after tool introduction in mid-January 2006. At the same time, coverage rates increased and the benefits described in Section 6 could be observed. Doing the same work, the developers realized (at least) tests with a higher coverage rate. This suggests that the quality of the work of the developers improved (cf. Section 5).

Over the whole development time the system had to cope with a high change rate. This was due to changing requirements and revised message type specifications originating from the external trading system.

Nagappan et al. [8] correlate a high change rate (churn) with a high probability of defects. At least in this project, we found the change rate metric to be less suited to hint at defects when compared to code coverage analysis and visualization or to non-commented source statements (NCSS). The reason may be a matter of scale – Nagappan examined a large system – or the system architecture. Here the architecture and the test strategy were especially designed to cope with frequent changes in certain, expected areas. The changes were expected (and occurred) in the plug-ins. However, a small change in the core could cause many defects in the plug-ins, whereas massive changes in a plug-in resulted in a minor number of defects.

8. Lessons Learned

In this section, we compiled our most prominent experiences and recommendations for the effective use of code coverage analysis and visualization tools. These experiences were not only made in the project covered in this report, but also in a couple of other projects in which the authors had been involved.

- Make expectations clear before introducing the tool. Given a reasonably usable automated test suite, make sure that the tool will mostly influence the developers to (in decreasing order) (1) write more robust code, (2) find bug or anomalies in the error handling, (3) find bugs in the handling of special cases in the blue-sky behavior, and (4) work on the consolidation of the test cases.
- Do not introduce coverage analysis and visualization tools in projects without a reasonably usable automated test suite.
- Do not expect to find many new bugs in the blue-sky behavior of your system. This is a direct consequence from the preceding lesson not to use these tools in projects without a reasonably usable automated test suite.
- Consider an introduction around mid-construction. From our experience, it is not necessary to operate this kind of tool over the complete development cycle. If introduction is too early, developers will waste time using a tool from which they cannot sufficiently benefit yet. If introduction is too late, though, the usage might be confined to the detection of bugs, which is only one of the benefits.
- Keep the feedback cycle between coding, testing and coverage visualization as short as possible. Developers will quickly lose the motivation to exploit coverage visualization if – in their opinion – the cognitive overhead to get the necessary information is too high. Therefore, adopt tools that integrate tightly with the development environment through visualization and annotations directly in the code editors.
- Make sure tool usage is well understood, esp. by junior developers, as they are the group that most likely takes the highest benefit out of it.
- Emphasize – over and over again – that the primary goal is not to reach an ultimately high coverage rate, but to exploit coverage visualization

for identifying areas of the code that are not covered by tests yet, but that are ‘test-worthy‘ from the (test) developers’ point of view.

- Award the identification of important untested areas and found bugs, but do not award high coverage rates. Resist the temptation to officially compete in the team or against other teams for high coverage rates.
- If you decide to prescribe a certain coverage rate or percentage, then prescribe a reasonably high one and make sure that it cannot be reached solely by testing the blue-sky behavior. If this coverage cannot be reached economically, confine this rate to certain important components.
- If you prescribe a certain coverage rate, be prepared that your developers will behave according to the metric, and unimportant parts will be tested only for the purpose to raise the coverage rate.

9. Conclusion

Code coverage and visualization tools can be an effective aid in enhancing software testing and improving the robustness of a system. In order to reap these benefits, our experience from a number of projects suggests that they must be properly introduced and judiciously used.

In this experience report, code coverage information gathered from one specific project was analyzed, and four specific questions concerning the impact and benefits of this approach were discussed. Based on this discussion, a list of ten lessons learned was derived. Abiding by the items of the list will help developers focus their use of code coverage tools. Failure to do so will likely result in an ineffective or in some cases even counterproductive use of the tools.

From our experience and the data from a couple of projects [1][13], most of the effort needed for test automation is strongly related to the testability of the system. When a system is not specifically designed for testability and a coverage analysis and visualization tool is used, the coverage rate often stalls at around 70 to 80%. Getting higher rates is often very expensive because the system’s architecture makes it almost impossible to cover certain parts of the code. Given a system with a reasonably testable architecture, the coverage of the tests will stall at a similar level, unless a code coverage analysis and visualization tool is used.

As future work, we aim to investigate the interrelationship between code coverage tools and both testable architectures and architectures with severe limitations in testability. Questions of theoretical and very practical interest alike will include: What is the influence of the testability of an architecture on code coverage levels? What is the effect on defect detection rates? Is there synergy between architecture and code coverage tools? Furthermore, the application of our lessons learned may have quite an impact on the overall testing process. Therefore, we plan to investigate the affected process facets, as well as the impact of the underlying software development process [11].

References

[1] Berner, S., R. Weber, and R. K. Keller. „Observations and Lessons Learned from Automated Testing”, *Proceedings of the 27th Intl. Conf. on Software Engineering*, pp. 571-579, St. Louis, MO, USA, May 2005. ACM Press.

[2] Clover, http://www.cenqua.com/clover

[3] Hutchins, M., H. Foster, T. Goradia, and T. J. Ostrand. “Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria”, *Proceedings of the 16th Intl. Conf. on Software Engineering*, pages 191-200, Sorrento, Italy, May 1994. ACM Press.

[4] Hansel & Gretel, http://hansel.sourceforge.net/

[5] JCover, http://www.codework.com/JCover/product.html

[6] http://www.junitee.org (JUnitEE)

[7] Lawrance, J., S. Clarke, M. Burnett, and G. Rothermel. “How Well Do Professional Developers Test with Code Coverage Visualizations? An Empirical Study”, *Proc. of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 53-60, Dallas, TX, USA, Sept. 2005.

[8] Nagappan, N. and T. Ball. “Use of Relative Code Churn Measures to Predict System Defect Density”, *Proceedings of the 27th Intl. Conf. on Software Engineering*, pp. 284-292, St. Louis, MO, USA, May 2005. ACM Press.

[9] Rational Application Developer for WebSphere Software, http://www-306.ibm.com/software/awdtools/developer/application/index.html

[10] Shaw, M. and Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[11] Talby, D, Keren, A., Hazzan, O., and Dubinsky, Y. “Agile Software Testing in a Large-Scale Project”, *IEEE Software*, 23(4), July/Aug. 2006, pages 30-37.

[12] VisualStudio, http://msdn.microsoft.com/vstudio/teamsystem/default.aspx

[13] Weber R., Th. Helfenberger, and R. K. Keller. „Fit for Change: Steps towards Effective Software Maintenance”, *Industrial and Tool Proceedings of the International Conference on Software Maintenance*, pp. 26-33, Budapest, Hungary, 2005. IEEE.

[14] http://en.wikipedia.org/wiki/Principle_of_least_astonishment