Delta Analysis

Nils Göde, Florian Deissenboeck

CQSE GmbH
Lichtenbergstr. 8, 85748 Garching bei München, Germany

{goede, deissenboeck}@cqse.eu

Abstract

We use various kinds of static analyses to identify problems that decrease the quality of our system. In many cases, however, the number of reported problems is huge—preventing us from solving these problems due to a lack of resources or motivation. We suggest a technique called "delta analysis" together with a simple behavioral rule that allows to deal with large numbers of problems and gradually improves the quality of our system.

1 Introduction

Improving the quality of our software is usually a two-step process. First, we have to identify existing problems and then, we have to remove them. Various kinds of static analyses exists to automatically detect and report existing problems in our software. But while the analyses are often fully automated and easy to apply, the second step—actually removing the problems—is a much harder task.

Independent from the specific type of analysis, tools are likely to report thousands of problems for real-world software which is usually long-lived and contains much legacy code. Since not all of these "problems" have to be real problems and many of them may be more of an indication, we use the more general term *finding* for the remainder of this paper. Especially if no quality control measures have been taken in the past, we often find ourselves confronted with a huge and unmanageable pile of findings when running an initial analysis. Removing all findings at once is in almost all cases infeasible, due to the limitation of resources and the risk of introducing new defects when removing existing findings. In addition, the huge number of findings itself and the comparatively small progress we make in removing them may decrease our motivation and make us resign.

In the remainder of this paper, we summarize our idea of an incremental improvement strategy and explain how it is supplemented by a technique named "delta analysis". The combination of both results in a continuous reduction of findings and ultimately leads to higher quality.

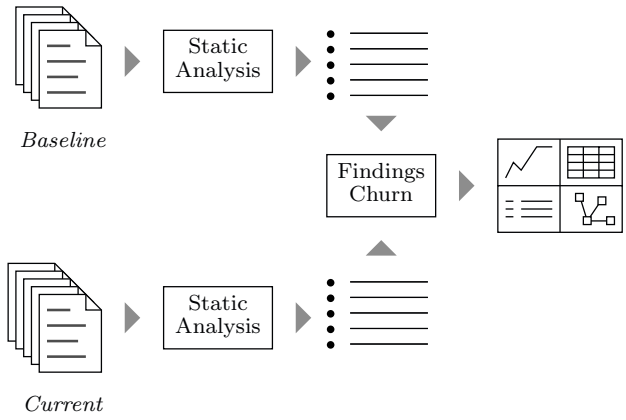

Figure 1: Delta Analysis

2 Incremental Improvement

Instead of solving all findings in a single session, which is in most cases infeasible, we suggest to incrementally remove findings. The central concept of our approach is *whenever you change a file, leave it in a better state than it was before*—inspired by Robert C. Martin's interpretation of the American Boy Scout rule *leave the campground cleaner than you found it* [5].

In our case "better state" or "cleaner" means less findings detected by static analyses. Our assumption is that when you have to change a file, you are forced to read and understand its contents. Given that you have read and understood the file, you are in a very good position to solve at least some of the other findings within that file. Furthermore, the original change and any additionally removed findings can be tested in a single run and do not require separate test runs. If the simple rule is obeyed in any case, the number of findings will naturally go down whereas the quality of our system improves.

On the one hand, we do deliberately not insist on leaving the file in a perfect state, because that may—depending on the number of findings within that file—result in developers not removing any findings due to limited time or a lack of motivation. On the other hand, we require at least a little improvement to ensure a gradual increase of the overall quality.

3 Delta Analysis

At the heart of our delta analysis is comparing the findings found within two successive snapshots of our system. We refer to the earlier snapshot as "baseline" as it defines the state of our system which we would like to improve. Findings within the baseline are seen as legacy problems for which we accept that they exist. We then execute the static analyses for both snapshots separately and compare the detected findings. The result is the *findings churn*, telling us about which findings have been removed, which findings remain, and which findings have been newly introduced. The findings churn allows us to decide whether the state of a file is better after it has been changed. We suggest to integrate the results into a quality dashboard to make them accessible to developers and project managers. The process is illustrated in Figure 1.

In comparison to the traditional static analysis with thousands of findings, we are now confronted with only a comparatively small and manageable number of findings. Furthermore, all findings are related to our latest activity as the delta analysis "hides" all legacy findings which already existed in the baseline. We can now combine this information with our rule of leaving each file in a better state than before to ensure that the overall quality or our system is gradually improving.

4 Advantages

One of the advantages of delta analysis is that the acceptance of the developers is high since they can chose how many findings they actually remove. They are not confronted with an unmanageable number of findings, but can use the results of the delta analysis to inspect those findings that are related to their recent activity. Furthermore, the overall progress of improving the system's quality is measurable by counting the number of removed findings. Being able to actually see that the number of findings is continuously reduced may also increase the developers' motivation and acceptance of quality control measures.

Another positive aspect of delta analysis is that it is not limited to specific types of analyses, but works for all kinds of findings. These include, for example, exceeded metric thresholds, architecture violations [2], and code clones [4]. The concept of delta analysis is independent from the actual source of findings and can also be used for findings detected by tools like *PMD* or *FindBugs*.

Delta analysis also allows to integrate new checks smoothly. Using simple static analyses, integrating new checks usually results in a huge number of findings in the initial analysis and all too often causes the abandonment of the new rule. Using delta analysis makes the integration of new checks much easier, because it focuses on those findings that were introduced since the baseline.

From a technical perspective, results are not stored in a database but the analyses are run for the baseline and the current snapshot of the system every time. This provides a great deal of flexibility, because the baseline can easily be changed. In addition, multiple baselines can be used to analyze the stepwise improvement of the system's quality. Running the delta analysis for the baseline and the current snapshot also allows to change the static analysis algorithms or their configuration while maintaining the comparability of the snapshots.

5 Experience

We have implemented the delta analysis using the ConQAT [1, 3] toolkit. We have integrated the delta analysis as part of the continuous quality control in about two dozen projects of two of our customers, *Munich RE* and *ABB*. The delta analysis has been used in these projects for some time and the feedback from the users is consistently positive. In particular, the users appreciate the possibility to focus on relevant findings which are related to their latest activity. Some of them even regard the delta analysis as an inevitable building block of quality control without which static analyses would not be applicable to large and long-lived systems.

6 Conclusion

In summary, we conclude that delta analysis is a crucial part of continuous quality control as it adds significant value to static analyses as our customers confirmed. Especially for long-lived systems, delta analysis helps to focus on relevant findings by abstracting from the unmanageably huge pile of legacy problems.

References

[1] ConQAT. `www.conqat.org`.

[2] F. Deissenboeck, L. Heinemann, B. Hummel, and E. Juergens. Flexible architecture conformance assessment with conqat. In *Proceedings of the International Conference on Software Engineering*, pages 247–250. ACM, 2010.

[3] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. Mas y Parareda, and M. Pizka. Tool support for continuous quality control. *IEEE Software*, 25(5):60–67, 2008.

[4] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings of the International Conference on Software Engineering*, pages 485–495. IEEE Computer Society, 2009.

[5] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.