Leveraging pre-commit hooks for context-sensitive checklists: a case study

Tobias Baum
Fachbereich Software Engineering
Leibniz Universität Hannover, Germany
tobias.baum@inf.uni-hannover.de

Abstract: The need to increase the efficiency of software development creates a demand for unobtrusive in-process means of software quality assurance that align well with agile processes. Many version control system clients provide facilities for running scripts or other executables before a change is committed to the repository. These "pre-commit hooks" can be client-side or server-side, both having distinct advantages. Client-side pre-commit hooks are well suited to implement context-sensitive checklists and automatic quality checks, but their potential has not been fully realized in practice and corresponding research is missing. To narrow this gap, we did an industrial case study on this use of client-side pre-commit hooks. The case study used the tool "TortoiseChecklist" and was focused on increasing continuous integration build stability. The results were positive, with a notable increase in stability and no significant disadvantages noticed.

1 Introduction

Many version control system clients, for example TortoiseSVN or the "git" command line client, provide facilities for running executables on the client computer before a change is committed to the repository. As these "pre-commit hooks" are called before committing, they can be used to check the change set and abort the commit on some condition. Being executed on the client side, they allow for user interaction, in contrast to so called "server-side" hooks that run on the repository server. Another feature of client-side pre-commit hooks (CPHs) is that they can be skipped more easily by a user. This has pros as well as cons: A pro is that the hook can be skipped, e.g. when it has an error that would otherwise keep an important fix from being committed to the repository. But on the downside, the checks in the CPH can be easily circumvented by single developers. Broadly spoken, our research interest is to investigate if CPHs can be used to improve the efficiency of the software development process. We did this case study to approach a small subset of this question.

The study took place in a medium sized German company that develops and sells a software product to external customers. The development team (13 developers) works co-located at the company's main site, using an agile process based on Scrum-ban. It uses a "homogeneous infrastructure" policy, where infrastructure and tools are provided centrally and should be the same for every developer. The team has been using continuous

integration (CI) based on Jenkins and static code analysis with Checkstyle and FindBugs for several years. One of the CI builds runs on every commit and checks unit tests and violations of static analysis rules. If there are violations or other problems, the build "fails" and every committer is notified. To further ensure stability of the integration builds, the team introduced a special role, whose task is to ensure that someone takes responsibility for fixing a failed build. The Eclipse plugins for static checks are used, too. Nevertheless, the stability of the build is quite low with only 64.5 % problem-free builds before the study.

Increasing the build stability would have a positive effect on development efficiency because there is less rework and context switching to do, there is less ambiguity in who is responsible for the failure and other developers are not negatively affected by instability in the common codebase. It is intuitively appealing that pre-commit checks can increase the build's stability. But not every check that is executed on the CI server can also be checked in a CPH, mostly due to time limitations. Over-reliance on CPHs leading to increased sloppiness could even lead to less stability. Therefore, the first research question (RQ1) is *"Do checks in CPHs increase the fraction of successful integration builds on the CI server?"*. Furthermore, there should be no negative side-effects from introducing CPHs. We want to study two possible side effects: A decreasing integration rate, and an excessive amount of time needed for the execution and checking of CPHs, resulting in the research questions RQ2 *"Do checks in CPHs lower the integration rate (commits per person and day)?"* and RQ3 *"How much time do checks in CPHs cost?"*. As CPHs are not mandatory, there is also the risk that they are often ignored, which would be a major disadvantage compared to server-side hooks. This leads to RQ4: *"How often are checks in CPHs skipped?"*

2 Related work

"RepoGuard"[LPR+09] and the system described in [Mer13] are similar in intention to TortoiseChecklist, but using server-side hooks instead of client-side hooks, leading to the tradeoffs discussed above. Context-sensitive checklists have been explored for design models in [Rob99]. Ståhl and Bosch[SB14] list common pre-integration procedures found in a literature review on continuous integration.

3 Details on TortoiseChecklist

TortoiseChecklist[1] is a client-side pre-commit hook for TortoiseSVN, a Subversion client. It evaluates checklist specifications, written in a JavaScript-based domain specific language, and presents the results in a GUI. The domain specific language can be extended by plugins, using Java's ServiceLoader mechanism. There is a common "global" checklist for all users of a repository, in addition to personal checklists for every user. It differentiates between two types of checklist items: "violations" and "questions". The former are used if it is highly likely that some item really denotes a defect, the latter are used if human judgment is needed and correspond more closely to the traditional use of "checklist item".

[1] available at https://github.com/tobiasbaum/TortoiseChecklist

4 Case study design

Data was collected before and after introducing the tool into the development process. Both collection intervals were chosen to have the same number of ”days with commits” (112 for each), resulting in a time range of 2013-11-01 until 2014-04-06 for ”before” and from 2014-05-08 until 2014-09-22 for ”after”. Between these intervals the tool was ”beta-tested” by us until it was introduced in a short training session and centrally activated for all developers. The global checklist contained all Checkstyle checks that were deemed feasible in a CPH (about 100 checks). FindBugs and further checklist questions were not activated by default but could be used in the personal checklists of the developers.

The following data was collected for the specific research questions:

RQ1: Number of failed builds and number of stable builds for the two time intervals. This data was extracted from the integration server's log files. Additionally, the number of commits having ”checkstyle” or ”findbugs” in their commit comment (called ”fix commits” for the rest of the paper) was measured. This number sheds more light onto the effect on lapsed static check violations. It was extracted from the SVN log.

RQ2: The number of commits that are not fix commits as well as the number of committers was measured. This was also extracted from the SVN log.

RQ3: The time that TortoiseChecklist was running was measured, which forms an upper bound on the time lost. Additionally, the time that its GUI was focused was measured, which forms a lower bound and better accounts for ”fixing with the checklist open”. Only calls that ended in a commit were taken into account, because there is no time ”lost” when the CPH found an error and the commit was therefore aborted. To gain these numbers, TortoiseChecklist was instrumented to log usage data to a network location.

RQ4: For every commit it was measured whether it was done with TortoiseChecklist active. These numbers were also extracted from the usage logs in combination with the SVN log.

5 Results

RQ1: The build stability increased from 64.5% (983 of 1523 builds) to 69.1% (1050 of 1519). This is statistically significant at the two-tailed 1% level (exact fisher-test, p=0.0079). The number of fix commits dropped sharply, from 9% (150 of 1660) to 5.5% (110 of 1987), which is also statistically significant (p=0.000057). Regarding RQ1 we can therefore conclude: CPHs can increase the stability of an integration build.

RQ2: The number of non-fix commits increased from 1510 to 1877, with 12 committers considered in the first and 13 in the second measurement interval (one developer joined in April). Accordingly the number of non-fix commits per person increased from 138 to 152, so that we could not observe a decreasing integration rate through the use of CPHs.

RQ3: The mean total time spent in TortoiseChecklist was 21 seconds (median: 6s), the

mean focused time was 10 seconds (median: 4s). This seems low enough to be viable in practice. There were large outliers, where the tool was left open for several hours.

RQ4: Analysis of the usage statistics showed that there are roughly three groups of developers: Some that used the tool for almost every commit from the start, some that began using it after some time and some that use it almost never. All in all, the tool was used for 69% of the commits (1377 of 1987).

As an industrial case study, there are a lot of threats to validity for these results, of which we can only describe the most significant here. Concerning internal validity, we clearly cannot rule out other influencing factors. Nevertheless, history and maturation bias seems unlikely: The number of developers and commits per build and average build duration did not decrease, the developers already had several years of experience with static analysis and CI and there were no other major changes during the study. To reduce bias resulting from the first author being a member of the development team, his contributions have been excluded from all statistics where this was possible. He also tried to refrain from advertising the tool use after its introduction, to reduce bias on the usage statistics. As a single-case study, we cannot make any claims regarding external validity, but we believe that the results should be generalizable to other cases that share some important characteristics: Similar usage of static analysis and CI, homogeneous version control infrastructure and a shared goal of producing high quality software in the team.

6 Conclusions and future work

Our case study showed the viability of using context-sensitive checklists based on client-side pre-commit hooks in an industrial context. Their use increased the stability of the integration server notably, without major disadvantages. Further studies shall research whether other positive effects can be obtained. Furthermore, we want to do qualitative investigation of the usage of the tool as a whole and of single features, especially the personal checklists.

References

[LPR+09] Malte Legenhausen, Stefan Pielicke, Jens Ruhmkorf, Heinrich Wendel, and Andreas Schreiber. RepoGuard: a framework for integration of development tools with source code repositories. In *Global Software Engineering, 2009. ICGSE 2009. Fourth IEEE International Conference on*, pages 328–331. IEEE, 2009.

[Mer13] Paulo Merson. Ultimate architecture enforcement: custom checks enforced at code-commit time. In *Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity*, pages 153–160. ACM, 2013.

[Rob99] Jason Elliot Robbins. *Cognitive support features for software development tools*. PhD thesis, University of California, Irvine, 1999.

[SB14] Daniel Ståhl and Jan Bosch. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87:48–59, 2014.