On the usage of Git hooks
Daniel Zaar [ tf08dz7@student.lth.se ]

Abstract—This paper started out as a naive idea that git hooks could be used as a main tool for continuous integration. In the end all it was was one implemented hook to help keep a clean repository. During this project the topic and contents of this paper was changed frequently and I had big problems getting started writing it – I liked the basic idea of its original thesis, but it never really blossomed to a proper paper. Instead the paper you are about to glance over is a summary of an experiment using a pre-commit hook during a agile software development project – where the coaching of said team got a lot more time then the creation of this paper.

The usage of a pre-commit hook can indeed help get a clean repository but it adds some requirements to the team. It is likely the worst paper I have ever written.

Index Terms—Clean repository, Git-hooks, Quality Assurance, eXtreme programming, five values, scripts

1 INTRODUCTION

As part of the examination of the course Coaching of programming teams given by Lund Institute of Technology the students are to dwelve deeper into a subject related to software development. The course is given in pararell with a course for second year students where they will develop a software project together using the agile methodology of eXtreme programming.

It is these students that will be the “coachees”.

In an agile (agile will henceforth be equivalent with eXtreme programming) development team every member has full access to every piece of source code i.e collective code ownership.

As one of the ways of spreading knowledge in the team[1] this can be seen as a part of Kent Beck’s values[2] (communication, simplicity, feedback, courage and respect, simplicity, and feedback.) This paper will explore how git hooks could be used to help “enforce” these values when working in such a team.

1.1 Hooks, you say?

The before mentioned Hooks are in fact executable script located in the .git folder of a repository. There are quite many of them that can executed in different stages of the versioning process. A selection of said hooks, for use on the client side, are listed below:

- **pre-commit** Invoked by git commit before making a commit.
- **commit-msg** Invoked by git commit, allows to inspect the commit message before allowing the commit.
- **pre-push** Invoked by git push before pushing to a remote reposotory
- **pre-rebase** Invoked by git rebase, can be used to prevent a branch from being re-based.
- **post-checkout** Invoked by git checkout. It cannot affect the outcome of a checkout but can be used to run validity checks of the repository,
- **post-merge** Invoked by git merge i.e when a git pull is done on a local repository. Cannot affect the outcome of a merge (won’t be executed if a merge failes due to conflicts)

These scripts don’t have any requirements more than being executable, i.e they are not bound to any specific programming language. The important thing is that the scripts returns a value of zero if successful and non-zero otherwise. Simply put, if the script “fails” i.e return a non-zero value the process connected to the failed hook will, in most cases, not proceed.

1. This is where all configuration of the repository lies
2. This hook is only available in git version > 1.8.2
All of the hooks listed above are one the client side of git and thus are run on the developers workstation. There are (of course) also hooks for use on the server side, i.e on the remote repository.

1.2 The git versioning tool

Although many versioning tools have some form of hooks I chose to use git, partly because I am more familiar with that specific tool but mostly because git has a wider selection of hooks than most other tools.

The git versioning tool was used in an centralized manor during this project, partly to keep it more familiar to the team, even though git is a distributed tool as its core.

1.3 The PVG project

The team will develop a race timing system for enduro racing during six iterations. Each iteration will represent a forty hour week i.e there will not be any development outside of these iterations. A iteration in this case last for one full day.

The main part of the learning goals for the “coachee” students is to learn the methodology of eXtreme programming and thus (preferable) follow Kent Beck’s values mentioned above. Each weeks iteration will focus on some particular parts of the methodology, for instance pair programming, test-driven development and clean repository.

1.4 Values

As mentioned above, the methodology of eXtreme programming is based on what Beck[2] called the four values, a sort of framework of guidelines to follow.

“We will be successful when we have a style that celebrates a consistent set of values that serve both human and commercial needs: communication, simplicity, feedback, and courage.”

Kent Beck

1.4.1 Communication

This is the first value for a reason – communication is vital for any team, an agile one as well. Several of the practices in XP are there to help and encourage the inter-team communication.

1.4.2 Feedback

Feedback about the current state of a system is important in an agile process, it feeds the iterations with information about how to improve for the next. It works on every timescale, unit tests, pair programming, planning process and acceptance testing.

1.4.3 Simplicity

“Keep it simple” is a common mantra in an agile team, not to be tempted to implement features that might be wanted in the future. More often than not, the requirements will change and the work will have been in vain.

1.4.4 Courage

Courage is last in this list, and perhaps not the easiest one to aid using git hooks.

2 Method

In the beginning, as this paper was merely an idea, I had hoped to use git hooks to drive some form of continuous integration. But as there was no practical way to make use of a local git server in order to make use of the hooks server side, the current form of this paper came to form.

Instead the choice fell on a private repository hosted by Bitbucket.org. As this is a “locked” server there is no way to implement any hooks of your own. Even though there are several hooks prepared for integration with jenkins, posting to a website or tweet new releases these were never explored during this project.

I had hoped to have a foundation of empirical data for this paper to stand on, but it proved to big a challenge to put in the time to create the scripts needed.
2.1 Implementing hooks
As the .git directory cannot be part of the repository the hooks is not under version control. This poses a problem as I cannot login to every students machine before each iteration and update all the scripts.

As the students all work in the Eclipse IDE, the solution was to create a folder in the repository but outside of the eclipse project and put all the hooks in this folder. I then made a small executable script that the team had to execute in order to symlink the hooks in the repository so that git can find them.

This allows me to edit and update the scripts and the changes will then propagate to the team as they pull from the repository.

2.1.1 pre-commit
To keep the repository clean the pre-commit hook was created as a script that simply ran all JUnit tests before allowing to commit. If a test failed the commit would not be allowed.

The idea with this was that the hook would act as a quality gate that would force the team to only commit “clean code that works”. Which it did – at first.

2.2 Problems and solutions
It did not take long before some team members ran into some problem with their code and decided to checkout a brand new repository. As they did that the deleted all contents of the .git folder and thus my hooks very rendered inactive.

It was soon clear that the team needed to follow some house rules to get this system working; as soon as the repository is reset they would need to run the provided script to link in the hooks from the repository to the correct place again.

Out of the five pairs in the team there was at least one that did not adhere to these rules. Either by calling git commit -n where the -n flag is used to bypass the hook and let the commit pass. This usally happend when the tests were all green but then failed due to new code being pulled down from the repository, i.e the new code wouldn’t integrate well with what the pair in question had written. Thus, excuses like “It isn’t our fault” or “but it worked just a minute ago” could be heard.

One big problem with this hook was on my end, my failure to manage to get the ant script to run all test in batch-mode:

```xml
<target name='test'>
  <batchtest>
    <fileset dir='${bin}'
      includes='**/Test*.class' />
  </batchtest>
</junit>
</target>
```

Instead I had to manually add new testcases as they were added which could easily have led to failing tests being commited to the repository despite the hook. Which in some sense renders the hook pointless. After iteration new classes was not created all that often and the problem was less severe.

Failure to adhere to the TDD practice in the team also led to some bad code being commited to the repository as there was no test to test it.

3 Results and conclusions
The main conclusion to be drawn from this experiment is that the creation of hooks takes time and skill. I have learned alot about hook and what they can be used for on a personal level, which is something.

So far I have put one hook to work in order to maintain a clean repository. Before every commit the pre-commit hook is run, the only thing it does is to run all the tests.

If there is a test that fails it pulls up a dialog window (since most students use eclipse and cannot see the console output) saying that a commit is not possible at this time due to failing tests. When all tests pass the commit will be allowed.

I went a bit back and forth between using the pre-commit or the pre-push hook but finally decided that the the push only pushes up the commits to the remote repository and thus the commit that ends up in the remote repository should all be valid, i.e all tests should always be green.
3.1 Teams opinions
The content of this section is based on oral discussions with the team during the weekly planning game.

- The script did stop at least a few commit due to failed tests each iteration.
- The team liked the idea of a guard to keep the repository clean
- The team had some complains about the hook making the commit process “to slow”

4 Discussion and Future work
The usage of git hook can indeed help the development process of a small software team. But it creates some extra requirements for the team in form of practices that need to be followed in order to make use of the hooks in place.

Using hooks also puts a requirement on the person implementing these scripts – He/she need to have a better knowledge than this author.

Git hooks could be used in a number of ways (although not empirically supported in this paper) to help compliance to some of the values. Communication

References