Winter Term 2011/2012

Scientific Paper

Distributed Code-Reviews

Using *Gerrit*

Patrick Permien, B.Sc., pp011@hdm-stuttgart.de
Computer Science And Media

Created for the lecture "Seminar Internet"
held by Prof. Dr. Simon Wiest
April 2, 2012

Abstract:
This paper gives an overview of the advantages and weaknesses of distributed sourcecode-review tools in software engineering. We cover this topic with a specific focus on Google's freely available software *Gerrit*. In chapter 1 we discuss how code-reviews are generally useful for groups of programmers. We lay out how traditional approaches differ from distributed setups where developers may be vastly distributed from a geographical point of view or where meetings are otherwise contraindicated. In chapter 2 we discuss how users can interact with Gerrit, and chapter 3 covers some basic knowledge for those people who have to administer one or more Gerrit installations. Finally, chapter 4 summarizes key points and gives an outlook on the future role of distributed code-review.

N.B.: This is an optimized version of the original paper. This paper has been modified with respect to the feedback from the supervising professor. The original version had been filed to Stuttgart Media University (Hochschule der Medien) on November 29, 2011.

Keywords:
Distributed Code-Review, Gerrit, Google, software engineering, source code, code quality, Git, DVCS, version control system, Jenkins, Hudson, programming workflow.

Contents

1 Motivation for Scheduling Code Reviews

1.1 Traditional Code Reviews

1.1.1 Purpose

For many years, recurrent reviews of source code have been established in software engineering as a common practice for ensuring code quality and constant knowledge transfer among developers. More specifically, code reviews are considered helpful for improving quality of source code by

- letting more developers have an eye on source code – similar to the peer-to-peer approach of pair programming;
- improving the odds of detecting bugs and weaknesses much earlier in the development and rollout processes, thus contributing to a structured approach to quality management;
- strengthen domain specific knowledge;
- triggering social effects of active collaboration, making the developers more of a team than singular contributors.

Literature exists on this topic in different variations and in multiple languages, including German. For an example, see Schatten et al. (2010), p. 117 ff.

It is important to note several aspects about the process of a code review:

- Code reviews are meant to happen in a non-threatening context.
- Code reviews should happen on a "line-by-line" critique fashion.
- Code reviews are meant to improve cooperation, not fault-finding.
- We are talking about voluntary code reviews, not the *involuntary* review of someone's broken code.

(van Rossum, 2006b, video at 1:17)

1.1.2 Expected Results of a Review

IEEE (1990) defines the term of a review in the context of software engineering as follows:

> "A process or meeting during which a work product, or set of work products, is presented to project personnel, managers, users, customers, or other interested parties for comment or approval. Types include code review, design review, formal qualification review, requirements review, test readiness review"

While highly formalized approaches are possible and useful, more informal reviews are common in daily work as well; those can be held completely internal.

In addition to the general review purposes discussed above, a single review should

- give quick feedback on specific programming artifacts that someone recently staged for discussion;
- make people think about their code in new ways rather than providing a concrete solution already.

Reviews are not intended to raise discussions on the general qualities of single developers. In practice, this principle is hard to maintain because quality of submitted work can (and should) quickly become obvious. (van Rossum, 2006a)

1.1.3 Contents of a Review

As indicated above, reviews should lead to discussions much rather than just criticism. Traditional review meetings support the approach of just pointing to problems rather than offering a solution, because otherwise the meeting is likely to last many hours. Meyer (2008) proposes the following nine standard sections of a review, ordered decreasingly by abstraction:

1. Choice of abstractions
2. Other aspects of API design
3. Other aspects of architecture (such as [...] inheritance hierarchies)
4. Contracts
5. Implementation, particularly the choice of data structures and algorithms
6. Programming style
7. Comments and documentation (including indexing/ note clauses)
8. Global comments
9. Adherence to official coding practices.

This pattern was already adopted for distributed reviews. Different approaches for traditional review settings are out of scope regarding the context of this paper.

1.2 Specialties of Distributed Code Reviews

As outlined by Meyer (2008), traditional reviews do in many cases not match with today's settings of development being distributed across locations, countries, or even timezones and continents. Problems arise around different working times, general trouble in intercultural communications, limited time and budgets for traveling, as well as technical limitations – such as remote workplaces communicating to the main centers possibly via a limited bandwidth VPN connection.

1.2.1 Popular Approaches

Many teams have opted for setting up a shared workbench toolkit for reviews, consisting for example of a VoIP client and any sort of document collaboration software. Some also include video or other additional features such as a shared virtual whiteboard depending on their needs. Meyer (2008) describes such a typical setup and shares some key experiences encountered with it.

1.2.2 Dedicated Tools for Organizing Code Reviews

The software *Gerrit*, that will be discussed in the following, comes as "a web based code review system, facilitating online code reviews for projects using the Git version control system" (cited from the project's index web page, `http://code.google.com/p/gerrit/`). For details on Git and its advantages, see Chacon (2009) and Gentz (2011).
Alternatives to Gerrit include the following software solutions:

- Google Mondrian – internal Google tool (van Rossum, 2006a)
- Atlassian Crucible – `http://www.atlassian.com/software/crucible/overview`
- Rietveld – `http://code.google.com/p/rietveld/`
- Review Board – `http://www.reviewboard.org/`

(Tang, 2011, S. 17)
Gerrit started as a set of patches to Rietveld, but quickly evolved into an independent fork. For details on the history of Gerrit, see the wiki article by The Gerrit developer community (2011a).

During my research I found another tool:

- CodeCollaborator
  `http://smartbear.com/products/development-tools/code-review/`

The various solutions differ in licensing, pricing, appearance and features. Yet they share the common idea of improving code quality with web-based reviews.

1.2.3 Comparison of Gerrit with Traditional Forms of Reviews

Gerrit characteristics: Gerrit is a web application written in Java and freely published under Apache License 2.0. It operates in a region- and timezone-indifferent way. Compared to VoIP or video conference sessions, Gerrit always implies a certain latency in response – hereby, we are of course referring much rather to the other developers involved than the equally inevitable network latency.
Distributed code review eliminates the need for scheduling meetings and the associated efforts for traveling to a conference or meeting room, etc.; it is also an advantage that the platform helps to stay focused on the actual content because it does not offer as much possibilities for distraction as a chat or physical meeting.

In many scenarios, the written form of feedback is much appreciated for documentary reasons and thus possibly favored over a session in e.g. Skype.
Gerrit requires teams to offer a definitive 'go-or-nogo' decision in traceable form (except when creating new branches or tags). However, this means that it restricts developers from contributing code that has not been reviewed yet. This strict policy may vary from many corporate environments where developers are trusted with commits while the review follows at a later spot. Differing from review meetings based on a fixed schedule, Gerrit works commit-based and hereby implicitly embraces a setup in which reviews refer to a unit of work instead of a unit of work time.

Visualization: Compared to some of its competitors, Gerrit currently offers a less fancy style. It is quite poor in inclusion or visualization of rich media content. Gerrit delivers core social web 2.0 characteristics in a very stripped-down presentation style. It will be interesting to see whether or not this will have a major impact on its future popularity. While the presentation can be changed with low effort[1], rich media content integration cannot easily be achieved.

GitHub Integration: With the Git community much aligning itself around the popular GitHub platform, the question arises whether or not it is a good idea to couple Gerrit with GitHub. A current web research showed that linking or coupling Gerrit with GitHub is generally not recommended by community of the developer forum *StackOverflow*. The StackOverflow community (2010) (namely S. Chacon) explains in detail why they consider this approach not helpful. In the first place, Gerrit already includes the server-side Git hosting, so coupling it with GitHub would cause redundancies.

It is beyond the scope of this paper to discuss the possibilities, advantages and trade-offs that come with a possible coupling of Gerrit with Github. Over time, best practices will evolve further and more research is likely to be initiated with respect to this approach.

[1] for an example of individualization, see https://review.typo3.org/

2 Gerrit for Users

2.1 Motivations to Use Gerrit

Like any other tool for distributed code reviews, Gerrit should be introduced in a development environment encapsulated in a communication strategy. Advertising Gerrit to the developers in advance will be a very important step in creating a non-hostile attitude among the team members towards the new tool.

This already implies that users do not necessarily choose to work with Gerrit voluntarily. They might be introduced to a project that already relies on Gerrit for reviews, or Gerrit may be introduced to a project they're working on without a consensus of all participating developers. Over time, there will be a significant number of users in a Gerrit system that did not choose Gerrit themselves. However, users can still bring their own motivations to use Gerrit.

Open Source Contributors: In an open source environment, chances are that a contributor will be inspired by a peer review from competent community members at eye level. In this case, he will feel respected and acknowledged for his skills and learning progress. He is likely to actively seek this kind of positive response. For a large number of detailed studies on the motivations and drivers of open source programmers, see for instance the list of works at `scholar.google.de/scholar?hl=de&q=open+source+motivation+programmer`.

Developers: In a corporate environment, the same motivation can apply but this is not necessarily the case. Young professionals will typically seek and enjoy the written approval by senior developers. In later years, however, code contributors might feel obligated to tackle another barrier after a long workday and find Gerrit to be hindering them in their daily workflows rather than seeing Gerrit as an inspiring knowledge exchange. The question is legitimate whether the strict rules of Gerrit are not overkill especially when it comes to trivial code changes.

Mentors: Finally, for project managers, supervisors or software consultants likewise, and for senior developers, Gerrit can indeed be a designated place in the workflow where they routinely post their suggestions of improvement. This can be a great help when they are in the position of a mentor or coach and Gerrit offers them a standardized way of giving feedback, replacing the need of giving such inputs via less suitable channels such as e-mail, lunchtime talk, or even VCS commit messages (i.e. fixing the issue themselves). The mentioned channels are less suitable because the feedback is much more likely to

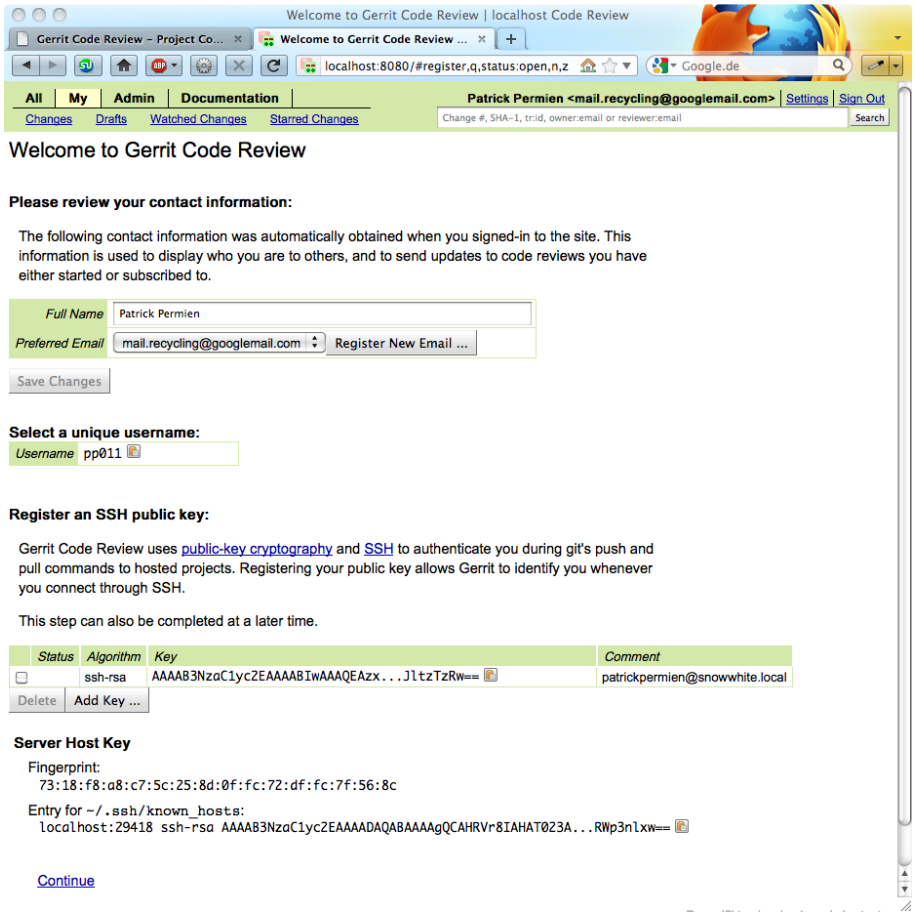

Figure 2.1: For developers who are comfortable with SSH and OpenID already, setting up a user account (depicted) should be relatively easy.

get lost, thus taking no effect of improvement, or causing a long tail of question-answer-correspondences.

Management: As laid out in section 1.1.2, the focus of code review sessions is to verify code quality much rather than the skills and level of a programmer. Yet of course code reviews will automatically become an indicator for management on who is among the top performers and who might fall short of standards.
Learning curves can be estimated when looking at review results of a contributor. It is also possible to check inside Gerrit how long a task has taken to be executed and by that means apply predictions to future work as to how long similar tasks might take. Gerrit does not explicitly support this approach by offering any statistics session, but the results can be derived manually from the raw data.

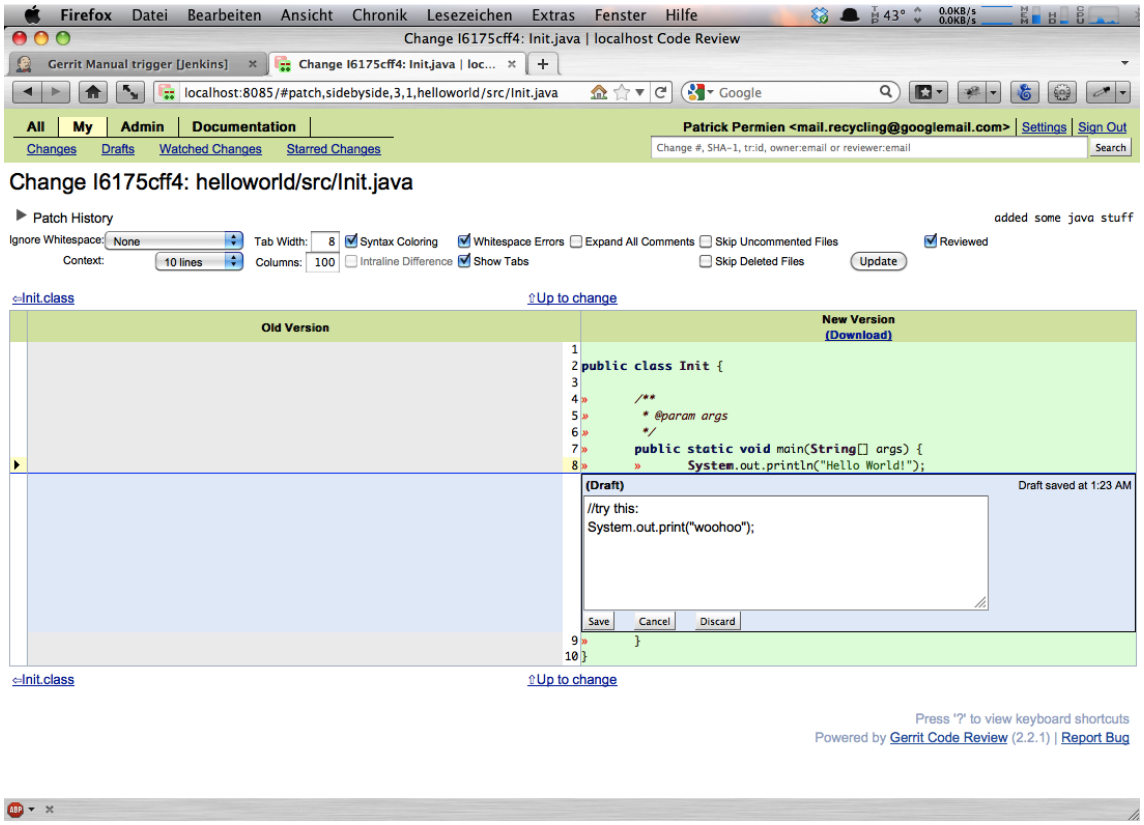

Figure 2.2: Gerrit offers the possibility to comment on the exact spot in the patch set that should be addressed

2.2 Workflows

The presentation by Aniszczyk (2011) describes the way Gerrit works in a handy overview that is cited in Figure 2.4. Gerrit comes to action only when the developer decides to push his local commits to the Gerrit server. This means that the Git-typical workflow of local development remains unaltered. Gerrit then hosts all the changes that developers push to it via SSH. The changes of a `push` are called `patch-set`.
The in-line comments offered by Gerrit offer a very comfortable fashion of annotating code. Contents can either be remarks expressed in prose or simply contain new source code suggestions.

Types of feedback: Gerrit differentiates between the confirmation "*label verified*" that confirms that the changes do compile and the confirmation "*code review*" which confirms that the code has been checked for its style etc. (criteria beyond severe errors). Every patch-set needs a certain level of such formal approval by other developers before it can be merged into the master. Which level that is must be decided by a policy setting in Gerrit. This defaults to one "verified" and two "code review" votes per patch-set.

If they encounter problems, a participating developer can flag the patch-set as *not* ready for merging and so express his vote for another review and alterations to the proposed changes.
The verification of a successful build can be created by a continuous integration (CI) service that builds the code including the patch-set and then checks the result for errors. Because the CI service (a Jenkins or Hudson instance) has access to Gerrit with a user account, the CI service can post that automated feedback like any other user can contribute to Gerrit reviews. A description of the CI concept is out of scope for this paper; literature is available, for example, see Duvall (2011) or Wiest (2011).

Possible outcomes: Changes may be approved, altered (and then re-submitted), or abandoned by the author. After a patch-set has been approved and merged with the master, the approved changes will be merged with the repository and Gerrit pushes the changes to the public Git repository that is hosted online. Optionally, CI may now come to action again: A Jenkins or Hudson server can start a build, triggered by the Gerrit event, if the CI service is configured.

2.2.1 Getting Gerrit to Work with Git

SSH: In order to be able to push changes from the local Git repository to the Gerrit review system, one must use Gerrit's dedicated SSH channel. By default, Gerrit's SSH runs on port 29418. Other SSH channels possibly offered by the server, like the system default, will not work with Gerrit since the necessary Gerrit commands are not available there.
Every user who desires to push to Gerrit must drop her own public SSH key on the server via the Gerrit web interface. It is associated with the Gerrit user name. The push command then looks somewhat similar to the following example:

Pushing changes to Gerrit

```
sleepingBeauty:ode patrickpermien\
        $ git push ssh://pp011@sleepingbeauty.local:29418/odeToGerrit
        HEAD:refs/for/patricksbranch1
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100\% (2/2), done.
Writing objects: 100\% (3/3), 356 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
remote:
remote: New Changes:
remote:   http://localhost:8080/2
remote:
To ssh://pp011@sleepingbeauty.local:29418/odeToGerrit
 * [new branch]      HEAD -> refs/for/patricksbranch1
```

In case of trouble, the Gerrit community is available via its mailing list that turned out to be vivid and helpful during the development of this work. A filed request by the author was answered within some two hours.

2.2.2 Best Practices

On an individual basis, many projects have established their own rules and standards which are to be followed in the Gerrit review process. However, in most cases these do not derive from any agreed-upon general standard associated with all Gerrit usages. Like with VCS commit messages, formal policies for the text contents can be rather strict or laissez-faire depending on the project's requirements and level of professionalism.

2.3 Issues

The top navigation of Gerrit's web front-end could be made easier to understand. The navigation links are organized in two levels and held very curt. It takes some time to find your way through.

Uncaught exceptions: While search options for changes are given in great detail, `NullPointerException`s happened to be thrown in the various test setups created for this paper every once in a while. This leads to a strong impression that the Gerrit software is not as "bullet-proof" yet as it could be. As another example, I did a port change from 8080 to 8081: Thereafter, in the front end all information appeared to be lost – but entering it again ended up with uncaught `duplicate key` errors from mySQL, indicating that it must still have been present in the underlying database. Of course, there are ways to work around such trouble. After all, a clean installation from scratch will do. Yet the software should prevent such basic errors from happening at all. Otherwise it will not serve the users by resolving issues but mostly create new ones of its own and can by this means fail its own purpose.

Gerrit's front-end error messages do in many cases not point out more than the plain error 500-equivalent output "Application Error – Server Error – Internal Server Error". It is then up to the administrator to look up the cause in the log files, leaving the (technically skilled) end user without a chance to get to know what caused the error. In another case, the test setup created the following error: "not a Gerrit project". As documented in the manual[1], this can ambiguously mean that either the project does in fact not exist or the user trying to access it has no access rights to do so. While this error text is a nice security precaution, it still misleads the user.

[1] http://gerrit-documentation.googlecode.com/svn/Documentation/2.2.0/error-not-a-gerrit-project.html

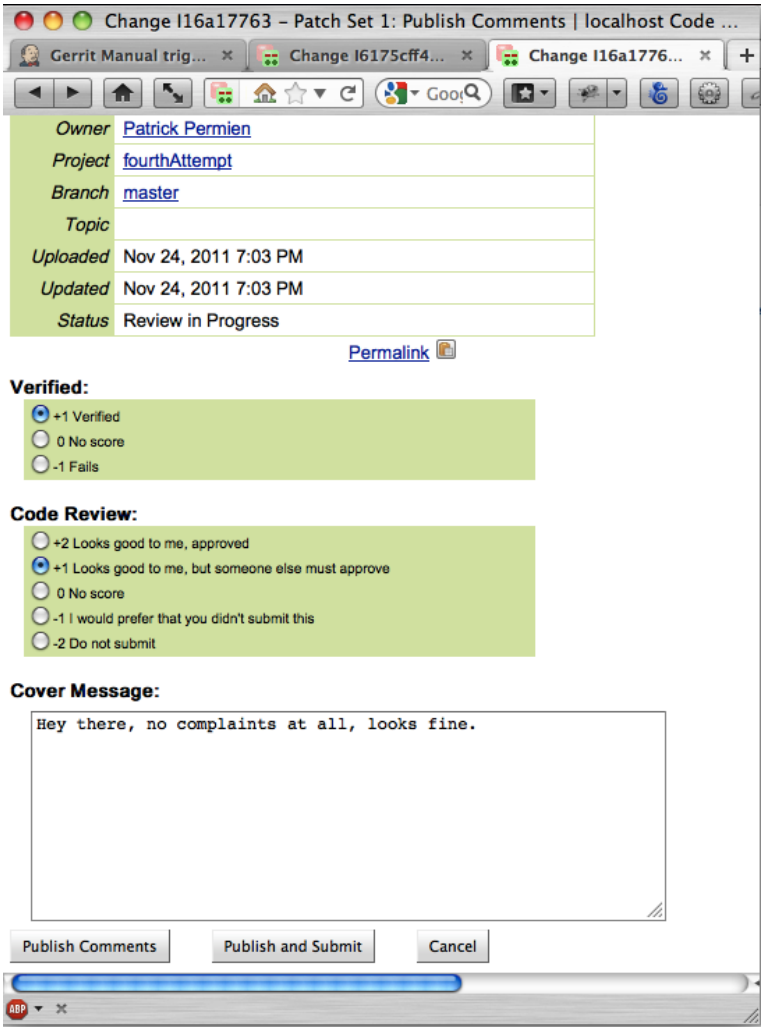

Figure 2.3: Users can be entitled to review other contributors' work and signalize their feedback as both rating flags and plain text.

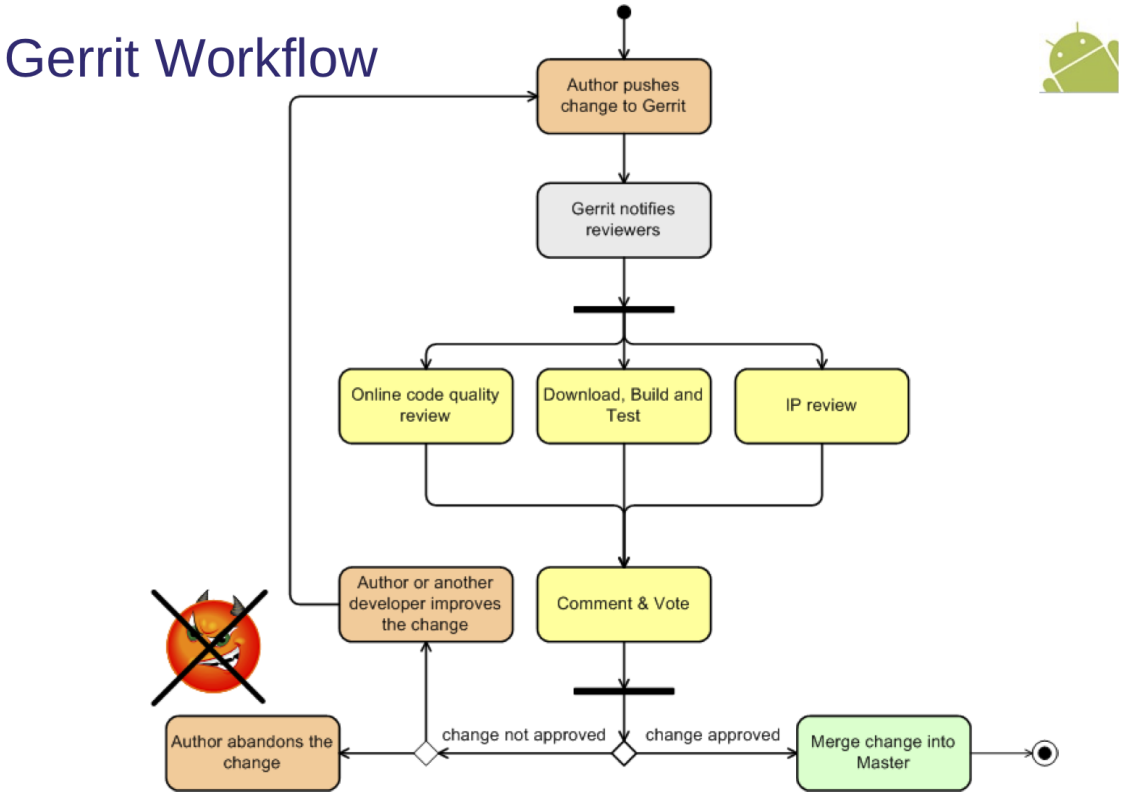

Figure 2.4: Gerrit workflow, as described by Aniszczyk

3 Administering Gerrit

3.1 Setting up Gerrit

3.1.1 Get Gerrit Up and Running on a Server

Installer: An installation package of Gerrit ships with an instance of the compact Jetty web server and is capable of delivering encrypted contents via HTTPS. Additionally, the installer includes several external tools that can be optionally downloaded during the installation process. The manual (The Gerrit developer community, 2011b) covers the installation of Gerrit on a server machine. The administrator must trigger the installer by executing the .war file with the `-init` parameter because a GUI is not available at present time.

Persistence: The administrator is asked to choose from the different supported databases. Gerrit can be configured to persist its data in either mySQL, H2 or postgreSQL. While in many corporate environments these may not generally be the databases of choice, it is at least possible to select a favorite option. It is important to note that the Gerrit database only holds the review data and the versioning still remains with Git at full extent.

Account administration: When administering large number of users, the option of signing in with open id is very helpful. However, an administrator needs to assign every single user to the respective groups individually.

Backup: It seems important to define a backup strategy ahead of runtime. In the test runs that were executed during the writing of this paper, Gerrit turned out to be somewhat fragile and crashed quite frequently. In a typical setup, Gerrit will quickly advance to a vital node in the development process, so data loss would imply that important documentation on decision making and collaboration goes missing. Gerrit comes with no backup mechanism of its own, so the administrator will have to include it in her existing canonical backup routines. The Git repository can be replicated to another server and of the database, dumps can routinely be created – the exact workflow then depends on the type of SQL database that is used with the Gerrit instance.
A more in-depth discussion on backup routines can be found in the Gerrit forum[1].

[1] http://groups.google.com/group/repo-discuss/browse_thread/thread/e466e8c60fa7337e/83508133d0bc9507?lnk=gst&q=backup#83508133d0bc9507

3.1.2 Start a Project

Creating a project: Prior to starting to work with Gerrit an administrator must create a new project within Gerrit. This is not currently possible via the web front-end and neither is a manual project setup. Instead, the administrator has to execute the command "`create-project`" on the Gerrit server machine. This is only possible whilst being connected to the Gerrit server via SSH. Gerrit's script will then do all the initial config automatically.

Further Configuration: Before being able to push to Gerrit via SSH, the admin must assign the associated rights to himself in the web interface, though. This is counter-intuitive. Like with the assignment of access rights, more project configuration can be done in the web front-end once the project has been created.

3.2 Setting Up Additional Services

The email notification service is setup during the installer routine. Gerrit asks for an SMTP server connection that it can use to send event notifications to users. The SMTP service can easily be delegated to reside on a different server machine.

CI: For a better integrated workflow, it is possible to connect Gerrit with the Jenkins/ Hudson continuous integration service. This coupling can be achieved on the Jenkins/ Hudson side by installing a trigger plugin for the CI server. Installing the plugin is quite intuitive and does not require too much work.

Details on Jenkins will be discussed in the upcoming class of this seminar. Subsequently, more in-depth information on continuous integration will be available within the context of this course. In literature, continuous integration is very well documented for various programming environments and consequently, CI is out of the scope of this paper.

You want to equip the Jenkins service with a user account of its own. This makes it easily traceable which of the response messages in the review cycle have been created automatically by the build process. For security and integrity reasons it can also be useful to set up a corresponding user group for bots, because it is then possible to assign an appropriate access rights vector to the automated contributors.

3.3 Recognize and Fix Problems

As of the time of this writing, it is hard to find any literature about distributed code reviews, let alone details on Gerrit. Troubleshooting must rely on the official documentation plus the general knowledge and sanity of a server admin who will have to look into `<gerritPath>/logs/error.log` for details. Many of the possible errors will be raised by standard components like SSH, the SMTP service, Jetty or Git. The usage of Console Log and Bash can be helpful because the installer writes log messages into the Console Log. In Bash, many commands offer a help output when called parametrized (`-h` or `--help`). SSH

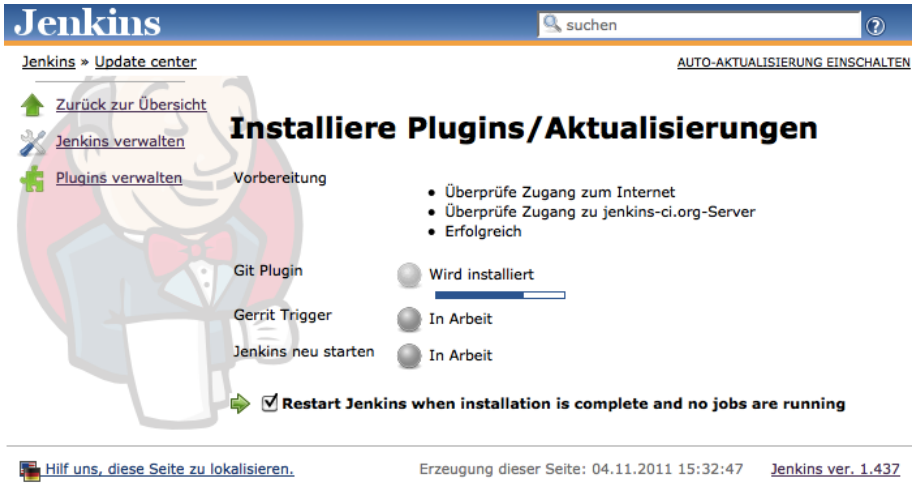

Figure 3.1: Installation of the Gerrit Trigger plugin on Jenkins

can be executed with enabled verbose output which will outline debug information to the requested level (`-v` / `-vv` / `-vvv`). If an error is really hard to fix, the mailing list may be of considerate help.

In some cases, version upgrades of Gerrit have required SQL schema updates. The necessary SQL migration scripts have been provided online for download by the Gerrit team.

In setup and appearance, Gerrit does not yet appear really mature. A server management UI, even a simple one, was dearly missed during the test setups because the long-lasting question-answer-loopback of the installer routine was very tiring.

4 Resumé

A core strength of Gerrit is its high cohesion with the powerful Git DVCS as well as the integration with Jenkins/ Hudson. Due to the tight coupling with Git, alternatives to Git are not provided.

While the developers keep the Git-provided ability to have their repository local and still can execute commits, Gerrit then adds an online dependency.

Not only does Gerrit work for the Android development that it was originally intended for, but there is a number of delighted developers who have become Gerrit users already. For an example, see Blewitt (2011).

Gerrit turns out to be useful for vivid and open communities. These attributes are prerequisites, given Gerrit's vital need for short response times in order to not cause frustration among contributors who want their code verified quickly. Gerrit also represents a nice option for a developer to remain rather anonymous compared to personal, peer-to-peer-style reviews or reviews in a group conference setting. In corporate usage scenarios, this will not likely be a relevant factor, but it is good for open source projects where for various reasons contributors might opt to not take credit for their work with their real names.

Possible problems may arise with rivaling developers or social malcontent: Social behaviors could turn up when contributors misuse the Gerrit platform to carry out their conflicts on a highly emotional level. This threat may be compared to the problem of "edit wars" in Wikipedia. Project leaders should be aware of such scenarios and think in beforehand about how likely this is to influence their daily routines in a negative way.

For easier integration into existing (especially corporate) IT landscapes, support for more database systems would be nice. In normal operation the administrator should have no need to manually adjust the database structure or modify its entries. The issue becomes more pressing when SQL migration scripts must be executed during version upgrades, as was the case with the introduction of v2.0.3.

Suitability: The author of this work considers Gerrit suitable especially for open source communities. When they opt to combine the advantages of using Git and its unique selling proposition of easy branching and merging with Gerrit, standardized procedures are easy to introduce.

Bibliography

Aniszczyk, C. (2011). An Introduction to Git and Gerrit (JUDCon Presentation Slides, Boston, MA), `https://www.jboss.org/dms/judcon/presentations/Boston2011/JUDConBoston2011_day1track2session6.pdf`.

Blewitt, A. (2011). Someday, `http://alblue.bandlem.com/2011/02/someday.html`.

Chacon, S. (2009). *Pro Git: Everything you need to know about the Git distributed source control tool*, Apress, New York, NY.

Duvall, P. (2011). *Continuous integration : improving software quality and reducing risk*, 6th edn, Addison-Wesley, Upper Saddle River, NJ; Munich.

Gentz, E. (2011). Die neue Freiheit bei der Versionskontrolle, `http://heise.de/-1224755`.

IEEE (1990). *610.12-1990 IEEE Standard Glossary of Software Engineering Terminology*, IEEE.

Meyer, B. (2008). Design and Code Reviews in the Age of the Internet, *Communications of the ACM* **51 (9)**: 86–71.

Schatten, A., Biffl, S., Demolsky, M., Gostischa-Franta, E., Oestreicher, T. and Winkler, D. (2010). *Best Practice Software-Engineering*, Spektrum Akademischer Verlag, Heidelberg.

Tang, M. (2011). Caesar: A Social Code Review Tool for Programming Education, `http://groups.csail.mit.edu/uid/other-pubs/masont-thesis.pdf`.

The Gerrit developer community (2011a). Background: The history behind Gerrit Code Review, `http://code.google.com/p/gerrit/wiki/Background`.

The Gerrit developer community (2011b). Gerrit Code Review v2.2.0 - Installation Guide, `http://gerrit-documentation.googlecode.com/svn/Documentation/2.2.0/install.html#customize`.

The StackOverflow community (2010). StackOverflow discussion: Gerrit with Github, `http://stackoverflow.com/questions/2451644/gerrit-with-github`.

van Rossum, G. (2006a). Mondrian: Code Review On The Web (Conference Slides), `http://rietveld.googlecode.com/files/Mondrian2006.pdf`.

van Rossum, G. (2006b). Mondrian: Code Review On The Web (Google Tech Talk), `http://www.youtube.com/watch?v=sMql3Di4Kgc`.

Wiest, S. G. (2011). *Continuous Integration mit Hudson: Grundlagen und Praxiswissen für Einsteiger und Umsteiger*, 1. aufl. edn, dpunkt-Verl., Heidelberg.
http://digitool.hbz-nrw.de:1801/webclient/DeliveryManager?pid=4028177&custom_att_2=simple_viewer

Remark: All internet sources cited here have been checked by the author of this work on April 2, 2012.