Chapter 1

Software Configuration Management and Continuous Integration

Matthias Molitor, 1856389

Reaching and maintaining a high quality level is essential for each today's software project. To accomplish this task, Software Configuration Management establishes processes and standards that support quality assurance.

In section 1.1 this summary gives an overview of the basics of Software Configuration Management. Afterwards an introduction into version control as one of its foundations is provided in section 1.2. As a way to monitor project health and increase quality even further, the technique of Continuous Integration is presented in section 1.3.

1.1 Software Configuration Management in General

SCM is about controlling the development process

A software system can be seen as a set of components that are combined to achieve a desired functionality. Software Configuration Management (SCM) is the task of controlling the development of these software fragments over time. It

supports project management as well as development and maintenance activities and tries to increase the overall quality of a software product [11].

Several activities in the SCM process ensure that the aimed goals are reached. Figure 1.1 shows these activities.

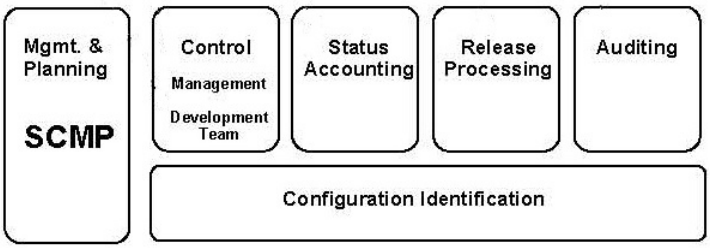

Figure 1.1: SCM Activities (based upon [11])

1.1.1 Software Configuration Management Plan

The SCMP provides a process framework

The software configuration management plan (SCMP) is the master plan of the SCM process. During software life cycle it is constantly maintained and updated if necessary. The plan defines responsibilities, mandatory policies as well as applicable procedures. It keeps schedules in mind and takes care of the available human and physical resources. Overall, the SCMP provides a framework for the whole development process [11].

1.1.2 Software Configuration Identification

Different configuration items must be identified

Software Configuration Identification builds the foundation for the other SCM activities. Its main task is the identification of different configuration items that have to be treated as an entity in the SCM process. Software configuration items are not just limited to code. Plans, specifications, documentation or used tools and 3rd party libraries are considered as items, too. It is also important to keep track of the relationship between the identified items, as this is

the basis to determine the impact of proposed changes later on. The set of configuration items at a selected point in time forms a baseline. Baselines are typically defined in the SCMP and may refer to a specific version (for example Release 1.0). Fixed baselines cannot be changed without a formal change control procedure [11].

1.1.3 Software Configuration Control

A procedure for change requests has to be defined

Software Configuration Control deals with managing changes that are applied to the software. It defines a process for change requests. An example of such a change request procedure is shown in Figure 1.2. First of all the

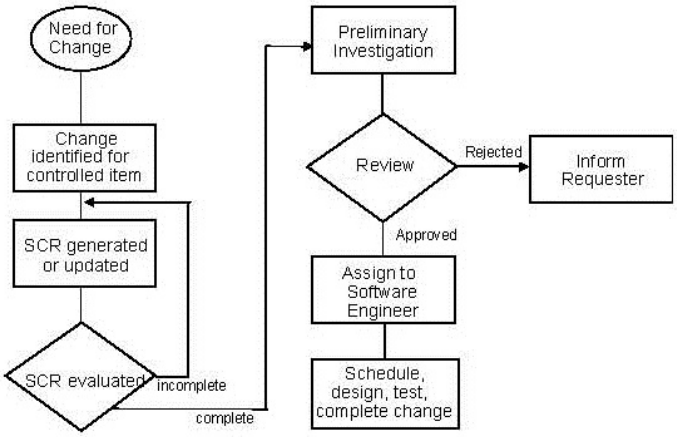

Figure 1.2: Example of a change request procedure (based upon [11])

need for a change is motivated. Afterwards the affected software configuration items are identified and a Software Change Request (SCR) is generated. Before the requested change is applied, it has to be reviewed and approved by a defined authority [11]. Tools like issue trackers may be used to keep track of the status of change requests and help to enforce the defined procedure. When implementing approved SCRs it is important to be able to map code or text changes to a specific change request. It is also required to know which baselines are affected by the applied changes [11]. Typically version control systems (VCS) are used to

document these changes.

1.1.4 Software Configuration Status Accounting

Project status reports support management

Software configuration status accounting is about recording and reporting the configuration status. These status updates are needed for effective management and may include information about the development process (for example the average time needed to implement a change request) as well as information about the software itself (for example average time between failures) [11]. Tracking project state and progress helps to identify upcoming problems early.

1.1.5 Software Release Management and Delivery

Release Management deals with building and distributing software

The release management activity concerns with building and distributing a software configuration. That does not only include delivery to the customer, but also intermediate steps like deployment of an application to a staging environment [11]. To ensure that the correct configuration items are picked for release and to avoid failures during deployment process, the release management should be automated [2]. A distribution plan makes sure that the software can be deployed anytime and that the deployment process is less error prone. As a result risks are reduced and confidence in the product increases.

1.1.6 Software Configuration Auditing

Audits are used to check conformance to conventions

Software configuration audits are performed to check the conformance of the configuration regarding functionality, standards and procedures [11]. Techniques like code reviews and static code analysis may me used to hold or support audits. Holding audits is important to ensure that defined standards and procedures are really applied.

1.2 Version Control

Version control supports Configuration Control and Status Accounting

Version control systems are essential for successful software configuration management. The use of these systems is the basic prerequisite for keeping track of content changes and, therefore, contributes directly to Configuration Control and Status Accounting.

Beyond that, version control systems provide further features that are required for professional software development.

1.2.1 Basic Vocabulary

Knowledge about the following terms is necessary when dealing with version control.

Definition: *Repository*

REPOSITORY:
A repository is a storage that contains the code of a project as well as its history [3].

Definition: *Working copy*

WORKING COPY:
The working copy is a view into the repository. It resides locally and contains all files that belong to the project. Code modifications are performed in the working copy [12].

Definition: *Commit*

COMMIT:
A commit is a set of changes that is send to a repository [3].

Definition: *Revision*

REVISION:
Each state of the repository that existed at some point in time is called a revision [3].

1.2.2 Common Features

A history of changes is recorded

History Log For each commit a message that describes the changes must be provided by the developer. These messages form the history log that can be consulted to find out why and when a specific change was introduced [9].

Figure 1.3 shows a part of an example history log.

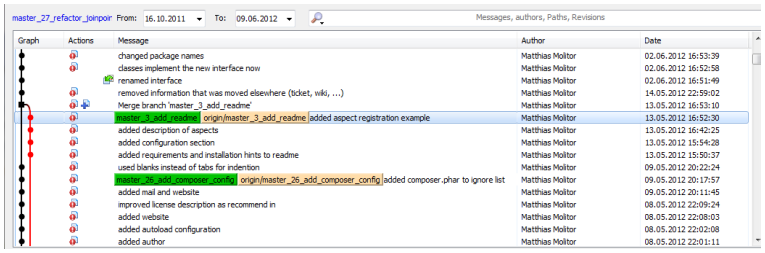

Figure 1.3: An example history log of a Git project

Different revisions can be compared

Diff of Changes Sometimes it is useful to check in detail what changes occurred. Therefore, version control systems are able to create so called diffs between files in different revisions. A diff shows the content in both versions and gives information about inserted, deleted and changed lines [9].

Figure 1.4 shows an example of a visualized diff.

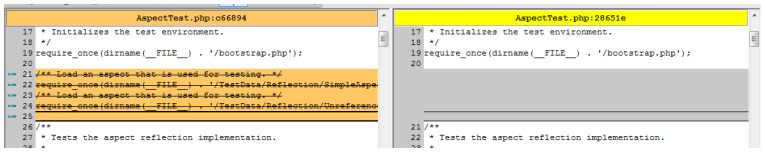

Figure 1.4: Example diff generated with TortoiseGit

Previous configuration states can be restored

Restoring Configurations Version control systems provide the ability to restore each software configuration that was committed. That allows developers to jump back in time and is especially useful if a change introduced unexpected problems.

Development lines can be separated

Branching To simplify parallel development, the concept of branching is supported. Any revision may be used to start a branch. Each branch represents a separate line of development that can be merged back later [9].

Figure 1.5 shows a branching situation. Branch `Feature` starts from `Main` in revision 1. Both branches advance in parallel until revision 2 of the `Feature` branch is merged back into `Main` again.

1.2.3 Evolution of Version Control

There are several version control systems that mark milestones in the evolution of software configuration management. Some of these are presented in the following sections.

1970s: SCCS provides basic version control features

Source Code Control System The Source Code Control System (SCCS) was developed in the 1970s at Bell Labs. It is used to manage revisions of text files. SCCS keeps a history of author, applied changes as well as reasons of a change on a per-file basis [7].

1980s: RCS simplifies version control

Revision Control System In the early 1980s the Revision Control System (RCS) was developed by Walter Tichy at Purdue University [8]. RCS is a set of unix commands that operate on single files. It keeps a revision history per file and supports common version control features such as branching and merging [13]. It runs completely locally and does not support a centralized repository [8].

1986: CVS introduces a remote repository

Concurrent Versions System The Concurrent Versions System (CVS) was created by Dick Grune in 1986. It builds upon RCS and adds support for projects with many files and multiple developers. CVS stores managed contents in a remote repository. It saves revisions of files instead of the state of the whole project, which causes several drawbacks.

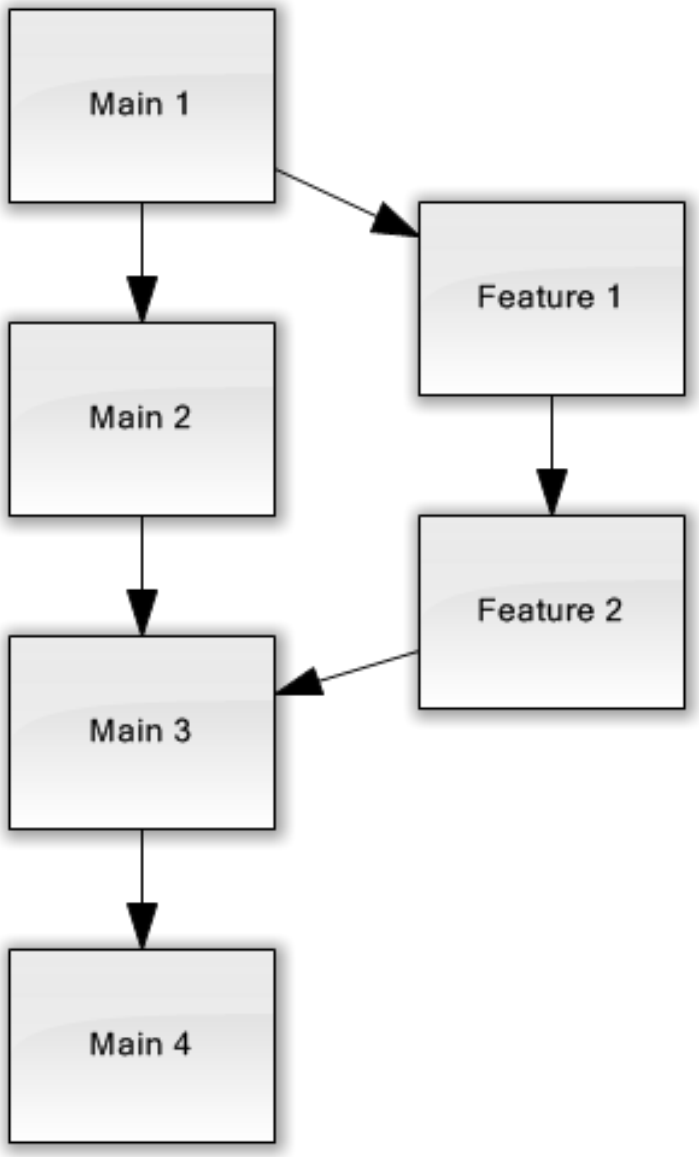

Figure 1.5: Example usage of branches

For example moving a file in the project leads to a loss of the previous file history. Also commits are not atomic, which may lead to an inconsistent state of the repository [8].

2000: SVN overcomes major issues of CVS

Subversion Subversion (known as SVN) startet in 2000 as an open source project that was initiated by the company CollabNet. It uses a central repository like the Concurrent Version System, but tries to overcome the major issues of CVS. Therefore, it does not store file revisions separately. Instead SVN is implemented as a kind of versioned file system that is able to follow changes within the file tree. This ensures that the history of moved files is not lost. SVN also maintains a project revision that is changed by each commit. Commits are are atomic which means that a commit is either fully applied or not accepted at all by the repository. Therefore, it is guaranteed the remote repository is always in a consistent state [3].

2005: Git starts to make distributed version control popular

Git Git was developed in 2005 as a tool to manage the development of the Linux kernel. Major requirements were speed and the ability to handle large projects efficiently. In contrast to CVS and SVN, this version control system does not rely on a central data repository. Instead, Git is fully distributed. Each working copy contains the full project history, including all revisions. Therefore, operations like committing, branching or viewing the project history can be performed locally, which makes them really fast compared to version control systems that rely on central repositories and require network connections. Sharing modifications is delayed in Git until explicitely requested by the user. Any Git repository may receive changes and is, therefore, a possible destination [1].

1.2.4 Current Version Control Systems

Currently the version control systems SVN and Git are very popular. The following sections give an introduction into the basic usage of both tools.

Subversion

SVN uses a central repository

As already mentioned SVN follows the central repository approach. Therefore, a repository must be created at server

side first.

A common convention is to use the following directory layout in the repository:

```
/trunk
/branches
/tags
```

The `trunk` directory contains the main development version. As the names indicate the other folders contain branches and tags [3].

Keeping these basics in mind, the console commands of Subversion will be used to maintain a local working copy.

To create a local working copy of the remote repository, the `checkout` command is used:

```
svn checkout http://example.com/repository/trunk .
```

This creates a working copy in the current directory.

After working with the contents in the working copy, the `status` command can be used to get a list of all changed items:

```
svn status
```

To put new files under version control, the `add` command must be used:

```
svn add readme.txt
```

Files that were not added will not be transmitted to the repository.

Before committing the modifications, it makes sense to check the repository for changes that occurred in the meanwhile and to merge these into the working copy. To accomplish this task, the `update` command is used:

```
svn update
```

Finally, the local changes can be committed to the remote repository:

```
svn commit -m "describe changes here"
```

Subversive integrates SVN into Eclipse

There are several tools that ease the day-to-day use of SVN. One of these tools is the Subversive plugin that integrates SVN into the widely used Eclipse IDE[1] . It can be installed easily via Eclipse Marketplace. The Marketplace can be found in the `Help` menu.

After installation the functions of the plugin are reachable from the `Team` menu (shown in Figure 1.6) that is accessible by right-clicking on the project or a project item. Additionally the Team Synchronizing perpective provides a quick overview of local and remote changes.

Git

Git does not require a central repository

In contrast to Subversion Git does not rely on a central repository. The distributed character of Git implicates that each working copy is also a repository.

Distributed version control establishes new working models

The absence of a centralized repository makes new working models possible. Figure 1.7 shows an example for the usage of distributed repositories with Git. Developers use the blessed repository as starting point, but they do not directly send back their changes. Instead, each developer sends changes to her own public repository. An integration manager is responsible for selecting changes from these repositories and merging them into the blessed repository.

The user identity is configured locally

When using Git the first time, the own identity that consists of a name and an email address should be configured:

```
git config --global user.name "My Name"
git config --global user.email "my.mail@address.com"
```

[1] http://www.eclipse.org/

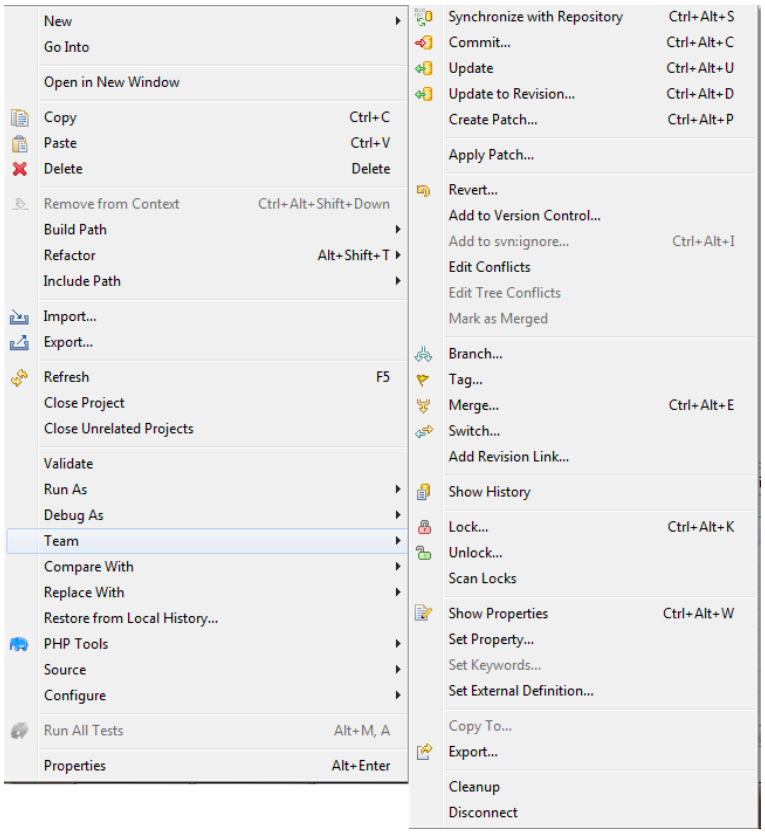

Figure 1.6: SVN integration into Eclipse

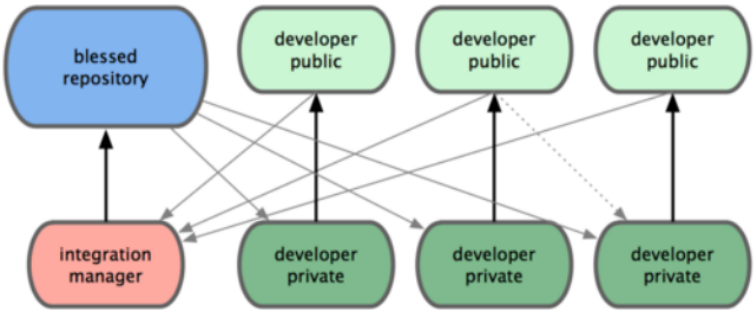

Figure 1.7: Usage of distributed repositories with Git [1]

There is no central server, therefore, each client has to provide its own identity.

SSH is required for authentication

Git uses SSH keys for authentication. A good tutorial for setting up the required keys can be found at Github[2] .

To create a local repository, the `init` command is used:

```
git init
```

This creates a new Git repository in the current directory.

When working with an existing repository, the first step is to create a local working copy via `clone`:

```
git clone git@github.com:geoquest/node_test.git
```

This command creates the folder `node_test` that contains the Git repository.

As in SVN, current changes are shown by the `status` command:

```
git status
```

The `add` command is used to register modifications for the next commit:

```
git add readme.txt
```

In contrast to Subversion, `add` does not put a file under version control in general, but it ensures that the current file changes will be included in the next commit. If the file is modified afterwards, `add` must be called again, otherwise the changes will not be committed.

The `commit` command looks the same as in SVN:

[2] https://help.github.com/articles/generating-ssh-keys

```
git commit -m "describe changes here"
```

But here another difference between Git and SVN is encountered: After committing changes in Git, the modifications are not immediately available in the remote repository. Instead, they exist only in the local repository and further operations are required to exchange data between the local copy and any remote repository.

To retrieve changes from a remote repository and merge these changes into the working copy, the `pull` command is used:

```
git pull origin master
```

The command above connects to the remote repository called `origin` and checks for changes in the `master` branch. If the working copy was cloned from a remote repository, then the repository `origin` is already configured and points to the repository that was used for cloning.

To send local changes to a remote repository, the `push` command is used:

```
git push origin master
```

This sends changes to the `master` branch in the `origin` repository.

Branching is extremely powerful in Git and helps keeping different lines of development separated. Additionally, managing branches works completely locally. To create a new branch, the `checkout` command with option `-b` can be used:

```
git checkout -b topic
```

This creates a new branch called `topic` and switches immediately to that branch. All modifications that are committed afterwards exist just in the `topic` branch.

After a while the changes must be merged back into the main branch:

```
git checkout master
git merge topic
```

First of all the target branch is selected. In this example the `checkout` command is used to switch to the `master` branch. Then the `merge` command is called to merge changes from the branch `topic` into the current branch.

EGit integrates Git into Eclipse

Just as for SVN, there is a plugin that integrates the functionality of Git into Eclipse. The addon EGit is also installable via Eclipse Marketplace and adds its functions to the Team menu that is shown in Figure 1.8.

SSH keys should be configured in Eclipse

As Git relies on SSH for authentication, it is important for EGit that the keys are configured properly in Eclipse. The menu that allows the key configuration is reachable via `Window ->Preferences ->General ->Network Connections ->SSH2` and is shown in Figure 1.9.

In case of error, EGit frequently blocks the message that is provided by Git. Therefore, it is often a good idea to switch to the console to investigate the situation if a problem occurs.

1.2.5 Comparison of VCS

Besides SVN and Git there are several other VCS available. Table 1.1 gives a rough overview of the features that are provided by some common VCS.

As there are so many different products, this list is neither complete, nor does it cover all features in depth.

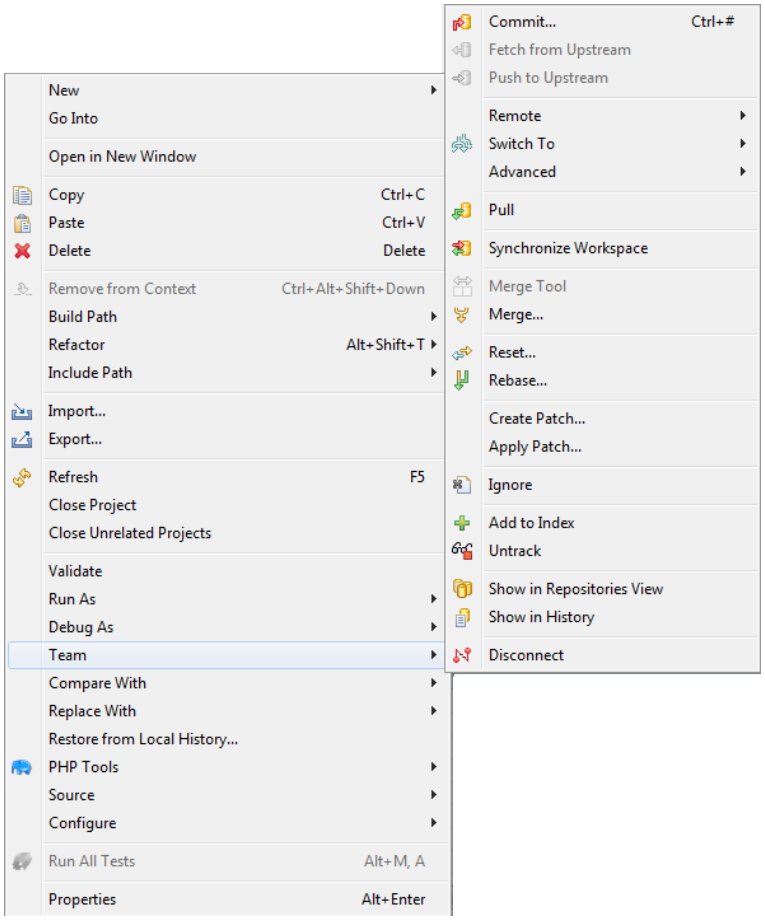

Figure 1.8: Git integration into Eclipse

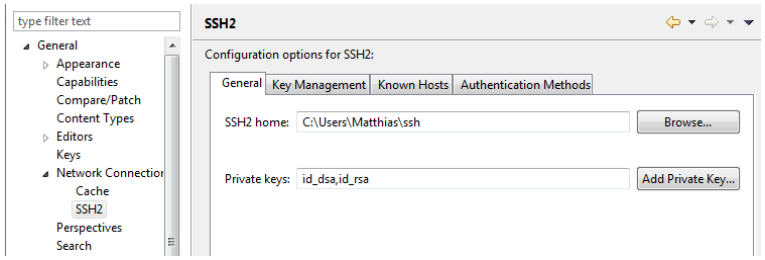

Figure 1.9: SSH key configuration in Eclipse

	SVN[a]	Git[b]	Perforce[c]	Bitkeeper[d]
Central repository	yes	no	yes	no
Atomic commits	yes	yes	yes	yes
Partial checkout possible	yes	no	unknown	unknown
Usable without network connection	no	yes	no	yes
Eclipse integration	available	available	available	not available
Web interface	available	available	available	unknown
Commercial	no	no	yes	yes

Table 1.1: Comparison of common VCS

[a] http://subversion.apache.org/
[b] http://git-scm.com/
[c] http://www.perforce.com/
[d] http://www.bitkeeper.com/

1.3 Continuous Integration

CI monitors project health

Continuous Integration (CI) adds an important value to SCM. By ensuring that each change is promptly integrated, it provides viable information about the health of a project.

CI integrates changes as early as possible

The basic idea is to integrate new parts of code as early as possible and to test the modifications in the context of the whole project. To achieve this goal, a CI server fetches the project source code after each change, compiles it and runs tests to check the functionality.

1.3.1 Requirements

To be able to use CI in a project, a few requirements must be fulfilled.

Location of current code must be known

Single Repository CI requires a central repository where the current code can be found. Therefore, it relies on the existence of a version control system that holds all the source that is required to build a project [6].

Building the software must not require manual steps

Automated builds A CI build must not contain any manual step. Thus it is essential that the project can be build automatically [5]. Build tools like Ant[3] or NAnt[4] can help to achieve this goal.

Code is tested against automated tests

Automated tests To be able to check the project status, a suite of automated tests is required [5]. Without tests, the CI system is limited to compiling the software. Testing frameworks like JUnit[5] are usually available for each programming language and simplify the creation of tests.

Builds run on a separate machine

Build machine To execute the CI builds, a reference machine is required. Such a dedicated system ensures that the builds are run under constant conditions and, therefore, makes the results comparable [4].

1.3.2 Advantages

By using CI, teams are rewarded with a couple of advantages that ease development.

CI notifies about failures early

Fast feedback When integration fails, team members are notified immediately. They can start to fix the problem while the number of performed changes is small and developers have the applied modifications still in mind [5].

Always being able to build increases quality

Increased quality Knowing to be able to build the software whenever necessary provides confidence into the developed product and increases overall quality [5].

Early integration reduces risks

Reduced risk Early integration avoids risking a long integration phase after finishing the development of all compo-

[3] http://ant.apache.org/
[4] http://nant.sourceforge.net/
[5] http://www.junit.org/

nents, because design failures that obstruct integration are discovered and fixed early [5].

CI contributes to status reporting

Improved reporting By providing additional information, CI supports the activity of Software Configuration Status Accounting. The gathered data ranges from a simple check, if the code is compilable, to complex metrics that are based on code analysis.

1.3.3 Rules

To get the most out of CI, there are a few rules each developer must follow.

Commit at least once a day

Frequent commits Changes must be commited frequently. Early integration does not provide a great benefit if the integrated modifications are already one month old. Therefore, each developer should commit her results at least once a day [5].

Keep the main development line stable

Stable development version The main development line is usually used as a starting point when creating new features and everybody should be able to consider it as stable. Therefore, obviously broken code should not be committed into the main branch [5].

Build local before committing

Local builds To avoid build failures, each developer should run a build locally before committing changes into the main branch [5].

Fix failed builds immediately

Immediately fixed builds When a build fails, the whole team should work together to fix it immediately. A build that always fails does not provide much useful information. Therefore, that state should not last longer than absolutely necessary [5].

Builds should run fast

Fast builds Builds must be fast to ensure that feedback is not unnecessarily delayed. It is recommended that a build does not take longer than 10 minutes [4].

1.3.4 Tools

Established tools simplify CI process

Although it is possible to apply CI without using 3rd party software, the usage of widely used and tested tools simplifies the process greatly.

The CI servers presented in the following sections are currently very popular and provide the framework to build a customized CI system.

Hudson/Jenkins

Hudson[6] is CI server that is written in Java and controlled by Oracle. Due to problems between Oracle and the Hudson community a fork named Jenkins[7] was created in 2011 [10].

As its community is much more active, the Jenkins server will be shown in the following. Nevertheless, the core functionality of Hudson and Jenkins is very similar.

Dashboard provides overview of projects

The dashboard of Jenkins is shown in Figure 1.10 and displays the configured projects. It provides a quick overview and highlights projects whose builds recently failed.

Project view provides more detailed information

By clicking on a project, its details page is shown. As seen in Figure 1.11 it provides a lot more detailed view on a specific project.

The build history is shown on the left. It tells the user when builds were executed and which of these builds were suc-

[6] http://www.hudson-ci.org/
[7] http://jenkins-ci.org/

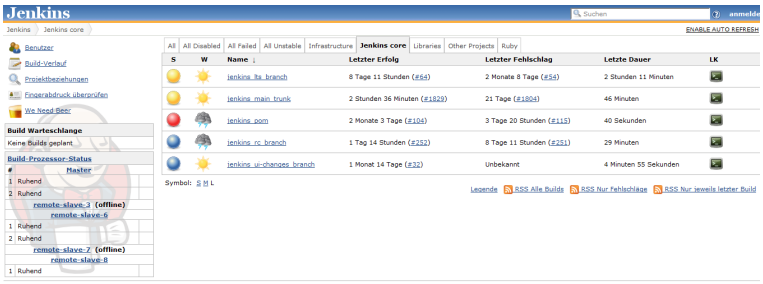

Figure 1.10: Dashboard of Jenkins CI

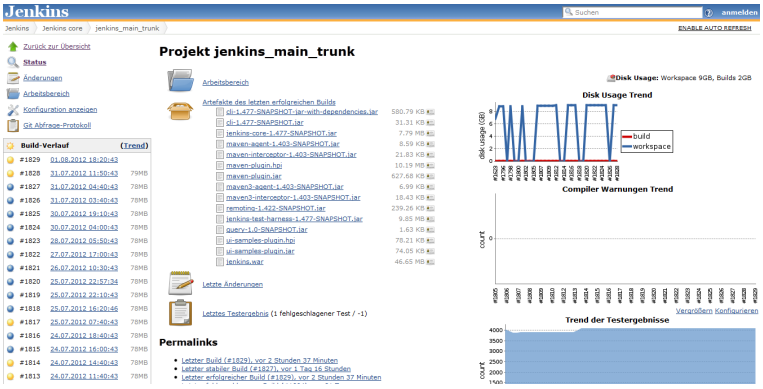

Figure 1.11: Project details view in Jenkins CI

cessful. By clicking on a specific build, additional information can be accessed.

On the right side several statistics are shown. This includes the number of tests as well as compiler errors and warnings over time. If available, more statistics can be added. For example advanced metrics that are gained by static code analysis are a viable addition to monitor the project status.

Travis CI

Travis provides CI as a service

Travis CI[8] is a rising CI system that is available as a service. It is tightly coupled to Github[9] and currently very famous among its users.

Builds can be executed in different environments easily

[8] http://travis-ci.org/
[9] https://github.com/

A big advantage of Travis CI is the fact that the same build can be executed in different environments, by just adding a line to a config file. Additionally, no dedicated CI server has to be provided and the service is free for open source projects.

Build is triggered when code is pushed to repository

To use Travis CI, the user has to sign in through her Github account first. Afterwards, the Travis CI profile page allows the user to activate CI for any of her projects. After activation, a build will be triggered whenever changes are pushed to the repository.

Build configuration is defined in `.travis.yml`

To tell Travis how to build the project, a config file named `.travis.yml` must be added to the repository root. The configuration contains the programming language that is used by the project as well as the versions that the software will be tested against. Details about the configuration possibilities can be found in the documentation[10] .

The build results are presented on the Travis website that is shown in Figure 1.12. Build failures and fixed builds are

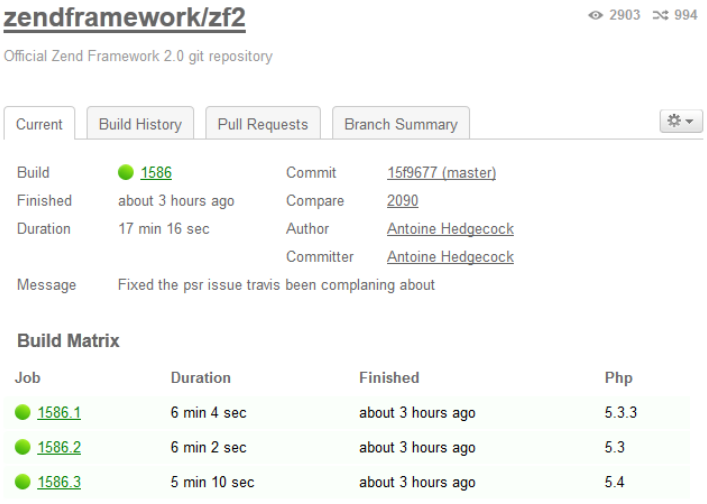

Figure 1.12: Build results in Travis

No support for creation of customized statistics

also reported via email.

[10] http://about.travis-ci.org/docs/user/getting-started/

It should be mentioned that the generated artifacts are removed when a build is finished. Therefore, Travis CI does not provide the means to create customized statistics over time without any difficulty.

1.4 Conclusion

This summary pointed out that SCM is about tracking changes and decisions as well as reporting and monitoring project status. As tools to track code changes, the widely used version control systems SVN and Git were presented and a short introduction was given. Building upon this technology, the method of Continuous Integration and its main principles were shown. The CI tools Jenkins and Travis CI were examined and an outlook on how these products contribute to reporting and monitoring was provided.

However, a summary cannot cover all of these topics in full depth. Further information about Subversion can be gained from [3]. The advanced usage of Git is explained in [12] and [1]. For Continuous Integration [5] can be used as a reference.

Bibliography

[1] S. Chacon. *Pro Git*. Apress, 2009.

[2] M. Clark. *Projekt-Automatisierung*. Carl Hanser Verlag München Wien, 2006.

[3] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato. *Versionskontrolle mit Subversion*. O'Reilly Verlag, 2005.

[4] P. Duvall. Automation for the people: Continuous Integration anti-patterns, 2007. Last accessed on 2012-08-02.

[5] P. M. Duvall, S. Matyas, and A. Glover. *Continuous Integration. Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.

[6] M. Fowler. Continuous Integration, 2006. Last accessed on 2012-08-01.

[7] A. L. Glasser. The evolution of a Source Code Control System. *SIGMETRICS Perform. Eval. Rev.*, 7(3-4):122–125, Jan. 1978.

[8] C. Henderson. *Building Scalable Web Sites*. O'Reilly Media Inc., first edition, 2006.

[9] A. Hunt and D. Thomas. *Der Pragmatische Programmierer*. Carl Hanser Verlag München Wien, 2003.

[10] A. Neumann. Erste Version des Jenkins-CI-Systems nach der Abspaltung von Oracle, 2011. Last accessed on 2012-08-02.

[11] J. A. Scott and D. Nisse, editors. *Guide to the Software Engineering Body of Knowledge*, chapter Software

Configuration Management. IEEE Computer Society Press, 2004.

[12] T. Swicegood. *Pragmatic Version Control Using Git*. Pragmatic Bookshelf, 2008.

[13] W. F. Tichy. RCS - A System for Version Control, 1985. Last accessed on 2012-07-28.

Typeset August 20, 2012