Software Configuration Management Plan

Jorne Laton, Sina Khakbaz Heshmati

November 5, 2009

Chapter 1

Configuration Management Planning

1.1 Abbreviations

In this document a number of abbreviations will be used, which are listed here:

SCMP Software Configuration Management Plan

SCM Software Configuration Management

CM Configuration Manager

CMA Configuration Manager Assistant

1.2 Configuration Tree

This is the structure which will be used to store all source files, reports, webpages and timesheets during the project. Every team member must be capable of choosing the right directory to commit new files. If this would not be the case, the CM or the CMA can be contacted to decide where certain new files should be added. If necessary, this structure can be changed to store a new type of file, but only members of the SCM are authorized to do so, other members are obliged to contact a member of the SCM to request such changes. For further explanation, see Chapter 2, Change Management.

1.2.1 trunk

This directory contains three important directories.

docs This is where all documentation about the project is stored.

utils Utilities can be found here. To give an example, Timetrack is a utility which is able to convert .xml-format files to other formats, such as .txt and .html. This utility is used to convert the timesheets, written in .xml, to .html, so that these can be placed on the website in a readable layout.

www The project itself: all source files, website files and libraries must be stored here.

1.2.2 branches

You can consider the project development as a timeline, in which the trunk is the main branch, but for each stable release, a new branch is added to the timeline. More information in Chapter 3, Version and Release.

1.2.3 tags

These are in fact 'snapshots' of the source code at a particular moment in time. When a stable version is achieved, it can be tagged. For example, a tag stable0.1 can be made to easily find this particular version. In other words, a tag is simply an other name for a revision.

1.2.4 timesheets

Each team member is responsible for his own timesheet, therefor everyone has to be able to write .xml-files, as the timesheets are to be committed in this format. The deadline for committing timesheets is set each Sunday, 23h59. To put these timesheets on the website, they are converted into html-format, using Timetrack, see Utilities. An example of this directory structure is:
timesheets/2009/week43/
In this directory the timesheets of week 43 of all team members are stored, but only in .xml-format.

1.2.5 minutes

Of each meeting a report is kept by the secretary, this report is made in LaTeX. This is saved in the appropriate directory, for each meeting a new directory is made. An example of this directory structure is:
minutes/2009/10/20/
This directory contains all necessary formats (.tex, .txt and .pdf) of the report of the meeting that took place on 20 October 2009. This date is also included in the filename of the report.

1.2.6 wiki

This is where wikis are stored. Those wikis can be manuals, agendas or other important documents for team members. They can easily be edited by every team member, be it via SVN or via Google Code itself. For example, a Requirements wiki was created to add ideas for requirements.

1.3 Subversion

Subversion has been chosen to regulate checkouts, commits and updates of project files. The arguments for this choice are the following:

Flexibility Misplaced adds or files with a badly chosen name, can be changed without problems, which is not the case in, for example, CVS, an older version based server system.

Revisions In SVN, commits are saved as revisions. In this way, all changes made to files are monitored very precisely, because each commit causes an increment of the revision index, which is then given to the new version of the file. Hence, it is possible to compare revisions.

Centralized SVN is created for use with centralized servers, which is ideal for project groups in which everyone knows and can contact everyone. No team member hosts the main directory, this is done by the central server. In a *decentralized* system the main software is hosted by the manager of the project.

1.4 Management Control

Every team member can checkout and update his working copy of the main software. To send back changes made to a file of the working copy, one has to commit this file. When no large changes are made, this should work without problems. The SCM strongly recommends to work incrementally on files, in other words, committing large changes on different places in a file is not encouraged. This is to avoid conflicts when trying to merge files edited by different programmers. More about this in the next section, Merging. If one, for example, wants to implement a new feature, which will drastically change the directory structure, he has to contact the SCM. The latter will consider this request, but more about this in Chapter 2, Change Management.

1.5 Merging

Imagine two members editing the same file, then committing it to the server at the same time. This will not be a problem, considering they both worked incrementally on the file, thus not committing a large change at once. Merging is what regulates this, by recognizing the changes of both sides it can create a new file containing both versions. It has to be kept in mind that this tool cannot perform miracles. If two members have worked on the exact same paragraph of a document and then commit, a conflict will occur. Merging will not provide a solution, but the last one who committed will be asked to clear out this conflict by comparing his version with the conflicting version.

1.6 Nightly Build

In the beginning this will not be dealt with, only once the source code is ready to be compiled and run. From then on, it will be a daily task to have runnable version of the code, which can be built at the end of the day. A day cannot be closed if the code is not compiled without errors, otherwise the next day would start with a non-runnable version which needs to be debugged first.

1.7 Website

Chapter 2

Change Management

Issues is the keyword in the Change Management, requesting changes happens via issues. An issue consists of two different sorts of variables: the status and the label(s).

2.1 Issue Status Values

2.1.1 Open

New An issue at this state has not yet received an initial review, it is pending, until the responsible management puts it in another state.

Accepted At this state, the issue has been acknowledged by the responsible management, its need is then confirmed.

Started Working on this issue has begun.

2.1.2 Closed

Fixed A developer has made the required changes to a source code file or a document. At this status it is up to the QA to check whether these changes really handled the problem.

Verified The QA has checked the fix and has confirmed that it is working.

Invalid This issue is rejected by the responsible management, it will not be implemented (yet).

Duplicate This issue report is a copy of or very similar to an already existing issue, therefor it is rejected.

WontFix It has been decided not to take action on this issue.

Done The requested task is completed.

2.2 Issue Labels

2.2.1 Type

Defect Reporting a bug in the software.

Enhancement Proposing a specific enhancement on the software.

Task Work item that does not change the source code or documents.

Review Request for a source code review.

Other An issue of a type that does not correspond to any of the above listed, will be given type-other.

2.2.2 Priority

Critical Must be resolved in the specified milestone.

High Want to be resolved in the specified milestone.

Medium Normal priority.

Low Does not necessarily need to be implemented in the current milestone, can slip in to later milestone.

2.2.3 OpSys

All Affects all operating systems (Windows, Linux and Mac OS X).

Windows, Linux or OSX Affects the respective operating system users.

Firefox, Internet Explorer, Safari Affects the respective browser users.

2.2.4 Milestone

Release0.1 Must-have requirements are implemented.

Release0.2 Want-to-have requirements are implemented.

Release1.0 All essential functionality implemented, tested and working. This ought to be a very stable version.

2.2.5 Component

UI Relates to program User Interface.

Logic Relates to application logic.

Persistence Relates to data storage components.

Utilities Utility and installation scripts.

Docs Relates to end-user documentation.

2.2.6 Others

Security Security risk to users.

Performance Program performance.

Usability Affects program usability.

Maintainability Hinders future changes.

Chapter 3

Version and Release

3.1 Milestones

The concept *milestone* is used to indicate a goal which is wanted to be achieved within a specified time. Parallel with the milestones are the releases, see 3.2 Release, and the branches, see 3.3 Branches.

3.2 Release

3.2.1 Releases

A *release* is in fact a runnable version of the project which includes all demands, according to the appropriate milestone. For each milestone, a release is built, this release is also a candidat to be the final working version, if no further releases are made. In this project there will be 2 to 3 milestones, hence also 2 or 3 releases will be made. The requirements are playing an important role in the release, especially the ordering of the requirements. The following milestones are set:

Milestone 1: The first milestone wants the software to include all must-have requirements, in order to be able to build a first release, which can be considered a possible solution for the assignment. It has to be noted that this first version is not at all a final version, because it contains only the most important features, no extras.

Milestone 2: In this stage all (or at least, most of) the want-to-have requirements have been implemented in the software. The corresponding release will be one which contains important extensions, so that the software has a much better usability.

Milestone 3: This should be the final stage, in which the nice-to-have requirements are added to the software. This version has a very good stability and usability.

3.2.2 Indexing

As said above, for each milestone, a release is created. These releases get specific indexes.

Release 0.1 This release corresponds with the first milestone, thus containing the must-haves.

Release 0.2 The second milestone, add of want-to-haves

Release 1.0 The final, stable version in which the nice-to-haves are included.

3.3 Branches

A *branch* is a directory with the same composition of the trunk. At the moment all must-haves are implemented in the source code in the trunk, the latter is copied to a new branch. This branch will be named release0_1. No further implementations will be made to this release, only fixing bugs will be done on this version. In the same way the second release will get its own branch, release0_2, and so on.

Chapter 4

System Build

4.1 Apache Ant

4.1.1 Compile

4.1.2 Javadoc

4.1.3 JUnit

4.1.4 JAr

4.1.5 Dist

4.1.6 Deploy

4.1.7 Run

4.2 Build names convention