American Journal of Software Engineering and Applications
2022; 11(2): 22-30
http://www.sciencepublishinggroup.com/j/ajsea
doi: 10.11648/j.ajsea.20221102.11
ISSN: 2327-2473 (Print); ISSN: 2327-249X (Online)

Expressions for Source Control Management Systems

Marco Schulz

Marco Schulz Consulting, Puebla, Mexico

Email address:
marco.schulz@outlook.com

To cite this article:
Marco Schulz. Expressions for Source Control Management Systems. *American Journal of Software Engineering and Applications*. Vol. 11, No. 2, 2022, pp. 22-30. doi: 10.11648/j.ajsea.20221102.11

Received: June 21, 2022; **Accepted**: July 8, 2022; **Published**: September 5, 2022

Abstract: In the last decades, many standards were established to increase productivity during Software Lifecycle Management. All these techniques and methodologies promise a higher success rate in software projects which could affirm themselves in the case the involved protagonists are willing to follow the instances recommended. Semantic Versioning, for example, addresses the information leak between functional changes, BugFixes and compatibility of existing and future releases of artifacts. Diving deeper into the daily craftsmanship of software projects enables us to identify the Source Control Management Systems (SCM) as a big treasure box. Much information can be extracted from these repositories, which are currently ignored for project analyzing. Expressions on SCM Commit Messages represent a new formalism that is both human-readable and machine-processable. Such a standard also forms a bridge between the code base and the requirements management and release management, since these activities are identified by a freely expandable vocabulary in the SCM. Another advantage of this strategy is the clear and compact expressiveness for development teams. A very practical aspect of my proposal is the easy applicability of the presented solution in real software development projects. As with the Semantic Versioning methodology already mentioned, there are no additional technical requirements to be met, since commit messages are a fundamental function of SCM systems. This paper discusses the option to improve data collection for controlling software projects and knowledge sharing in collaborative teams.

Keywords: Source Control Management (SCM), Configuration Management, Software Lifecycle Management (SLM), Software Engineering, Distributed Development, Team Collaboration, Software Maintenance

I. Introduction

Thinking about SCM systems we have to keep in mind, that since the first roll out of CVS in the early 1990‘s and today, many things have changed. Searching the free online encyclopedia Wikipedia, presents a page “Comparison of Version Control Software” which contains an overview of version control software of more than 30 SCM tools. This gives an idea why software companies usually have around three or more different SCM systems in work - of course the real amount depends on how many years they are in business.

The possibility to attach every revision in SCM Systems with a commit message allows the developer to inform other users with a short explanation of his work. This feature is extremely helpful by browsing the history manually in search of special code changes. If these commit messages well structured there exist a possibility to grab automated information of project growth. In this paper on expressions is introduced as solution for structured commit messages which could processed by software and also helps developers to resume their work more efficient.

The list of research on SCM is quite overwhelming and covers multiple aspects. The work of Walter F. Tichy on RCS [2] presents a deep fundamental insight into technical aspects of SCM systems. Abdullah Uz Tansel et al. gives in his research a brief history and builds a bridge to nowadays SCM systems [11]. The paper of Christian Bird et al. describes the ideas why companies deal with various SCM solutions [12]. Many existing papers like the one from Filip Van Rysselberghe and Serge Demeyer already identified SCM repositories as a significant information storage [5], which contains more than a simple history of source code. The approach from Louis Glassy to observe the growth of students in the software development process by using SCM techniques [6] demonstrates another method to grab implicit information from SCM. Alongside the fundamental research in software engineering, there exists a great resource of Blogs, articles and

books from people who are directly involved in the topic. They describe experiences and best practice to make the next release come true, as referred towards the web resources in the footnotes. A small selection of related practitioners books is also included in the reference list.

Let us take a closer look at how processes for SCM could be improved. For this reason, section II defines the terminology of this paper and talks in detail about merging and branching strategies. Section III remind some basic knowledge on SCM and gives a simple idea about how complex build and deploy pipelines interact. Following this quick journey, section IV draws a picture about real problems that occur in software development projects and explains possible Points of Interest (POI) inside an SCM repository. These fundamentals allow a definition of the vocabulary we introduce in section V. A real world example will demonstrate in VI the cardinality of the expression and gives ideas about its usage. After all, section VII will reflect and summarize these thoughts. The last section talks about ideas how future work could be continued.

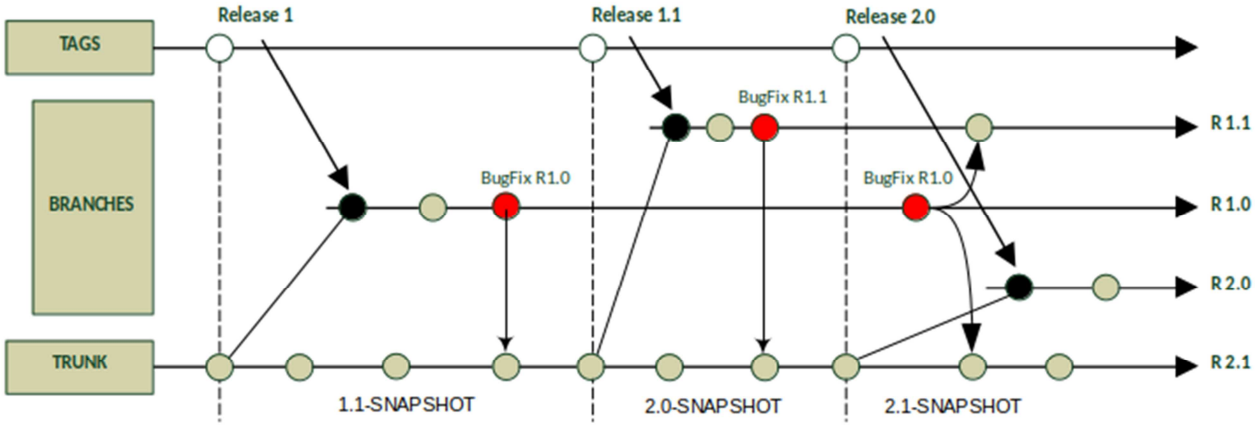

Figure 1. *Branch and Merge.*

2. Definitions

The definitions in this section are based on the English dictionary Merriam Webster with a contextual relation to SCM systems. The term Source Control Management System (SCM) is applied in this paper to describe tools like CVS, Subversion (SVN) or Git. Many other names have appeared over the years in literature for this type of tools. All these terms like Version Control System (VCS) or Revision Control System (RCS) are considered as equal to each other.

Artifact “A USUALLY SIMPLE OBJECT (SUCH AS A TOOL OR ORNAMENT) SHOWING HUMAN WORKMANSHIP OR MODIFICATION AS DISTINGUISHED FROM A NATURAL”.

OBJECT; “ESPECIALLY: AN OBJECT REMAINING FROM A PARTICULAR PERIOD”. In the context of SCM, an artifact is a binary result of the build process. Artifacts can be libraries, applications and so on.

Repository “A PLACE, ROOM, OR CONTAINER WHERE IS DEPOSITED OR STORED”. In software engineering a repository denotes a managed storage. We can distinguish repositories for source code and for binary artifacts.

Revision “A CHANGE OR A SET OF CHANGES THAT CORRECTS OR IMPROVES SOMETHING”. Each successful commit from a user to the SCM represents a change of the internal state in the SCM. These different states are revisions. Subversion for example increments an internal number after each commit [18]. This unique identifier is called revision number. Git on the other hand manages the revision number smarter and creates SHA-1 Hashes from each commit as an identifier [15]. This brings more flexibility for dealing with branches.

Release “TO GIVE PERMISSION FOR PUBLICATION, PERFORMANCE, EXHIBITION, OR SALE OF; ALSO: TO MAKE AVAILABLE TO THE PUBLIC”. A release defines a set of functional assertions for an artifact. When all functions are implemented, a test procedure is started to exclude as many failures as possible. After the termination of testing and corrections, the artifact gets packed for delivery. To distinguish the different versions of an artifact, it gets labeled by a unique version number. By convention, it is not allowed to have more than one artifact with the same version number.

Tag “A DESCRIPTIVE OR IDENTIFYING EPITHET”. - A Tag is a label to a special revision, like a release, and is used as bookmark.

Trunk “THE CENTRAL PART OF ANYTHING”. A trunk is a common convention and means the main branch, where the current development happens [17]. In Git this branch is called master for the local repository and orgin in the remote repository. Branching and Merging is one of the major feature in SCM systems and also a high sophisticated operation. It is not so unusual that developers and also Configuration Managers struggle with this. The paper of Shaun Phillips et al. contains a developer comment about the dealing with SCM and the pain of merging [10].

“We are a team of four senior developers (by which I mean we’re all over 40 with 20+ years each of development

experience) and not one of us has had a positive experience in the past with branching the mainline... The branch is easy - it's the merge at the end that's painful."

This shows that even persons with many years of experience need a detailed explanation of a seemingly trivial procedure. A simple understanding how branches typically have to be used and how they represent the evolution of a real software project is of high relevance for this paper. Figure 1 explains the optimal interaction between branches and the trunk which is described by Chuck Walrad and Darrel Strom as Branch by Release Model [3]. In addition to the context of branching and merging there is a version tree sample graph explained by Yongchang Ren et al. in their paper [8].

In order to give a comprehensive explanation of the process we assume a simple Java library project. As build tool Apache Maven is chosen which is successfully used for years by many different commercial and Open Source projects. Maven defines many standards for the software development process and implements them. Its success feature is a highly efficient dependency management.

The information about the artifact version number is managed in the pom.xml, the Maven build file. For this reason the POM has our special attention. In the context of Maven a versions number is labeled SNAPSHOT while it is still under development. This convention allows in collaborative teams the sharing of non official published artifacts. After removing the label SNAPSHOT the artifact is released. By convention it is not possible to have more than one artifact with the same version number. In section III this topic is discussed in more detail. For the moment it is necessary to know that this convention takes effect in collaborative processes. The correct way to share artifacts is the usage of a Repository Manager. The most common Repository Manager is Sonatype Nexus OSS which is used for Maven Central [19] to deliver dependencies. Nexus will refuse the request if a developer tries to publish an already existing release of an artifact. With this infrastructure it is not necessary to transfer binary artifacts to the SCM. This tool chain is a simple example for a highly complex infrastructure to build and deliver software in large companies.

In figure 1 the development starts with version 1.0-SNAPSHOT. After the release of this version, the development of the next version 1.1-SNAPSHOT continues in trunk. The revision of the released version 1.0 gets branched to fix some bugs. The branch will not be created automatically during the release, rather it gets created when there is a need, for example BugFixes. The branch will be named by its minor version 1.0 to stay flexible for further corrections. After a correct BugFix the changes get merged back to trunk and so on. It is very important to keep in mind, that after a release, no new functionality can be added to the versions 1.0.X, only corrections are allowed.

The merging of failure corrections can lead to complications if there already exist deployed versions. When a bug is detected down to an existing version it will be necessary to fix all following versions and increment their version number as part of the correction. For example if there exist released versions 1.0.2, 1.1.1, 1.2.3 & 2.0.1. and the fix has been done in version 1.0.2 it will have to be renamed 1.0.3 for release. The merge direction is always from the lower to the higher version which means that the version numbers of all following involved artifacts have to be increased. By this it can be assured that only fixes will be exchanged and no functionality is moving form an higher to a lower version within the merging process.

In this model the case of parallel feature development is missing. This happens when a very complex functionality is planned and the implementation cannot be finished in one release cycle. This especially often occurs in agile projects with a short time line between releases. Feature Branches address this requirement as well. The process is a simple extension of the Branch by Release Model. The Feature Branch will be created from the trunk and will be named like the feature. To test compatibility this branch at least needs to be merged from the trunk after each release. A merge can also be performed if the trunk provides important new features – whenever necessary.

A very useful advanced usage of branches is the stash command, that comes as build-in with Git. Indeed this feature is not so common but simple and powerful. Imagine a developer is working on some implementation with the urgency of having to deliver a BugFix for another release. He needs to switch his workspace to this branch but the current work needs to be saved without a direct commit to the trunk. The solution is create a branch and check in the current work and hence switch the branch for the fix. After all is done he will have to switch to the stashed branch, finish the work and merge the result to the trunk. An often observed procedure for developers are simultaneous checkouts of different branches and just switching the IDE workspace. By experience in large companies, this is very time consuming and error prone. By the law of Murphy, the only needed branch is the one not present in a local checkout collection.

To get in touch with branch models more profoundly, the website of the Git SCM [20] presents different branching workflows. Also at [21] exists a very detailed explanation for Git branch and merge best practices.

3. Quick Survey on SCM Basics

As described, there exists a huge amount of Source Control Management solutions. Even just picking out the most popular systems, we are able to identify many differences in detail. These may be the reasons why some tools have become more popular than others. Naturally, all of these systems do the job and are based on common ideas. A very early and fundamental work on SCM systems done by Tichy gives a deep insight about the Theory on how an SCM should be constructed [2]. Today, based on the approach of how things are done, we can classify them. Directory and file based systems, like Microsoft Visual Source Safe, are part of the less effective group of SCM. In commercial environments this group has low relevance

because quite often it causes inconsistencies of the repository. This leads us to the category of Client-Server solutions. Client-Server SCM systems have two manifestations: centralizedand distributed. SVN is the most famous representative for centralized solutions. In new projects the choice of the day will very often be Git, a very popular distributed SCM tool. In “Transition from Centralized to Decentralized Version Control Systems” the authors describes why decentralized SCM systems are favored by developers [12]. Interviews of developers have shown the benefits and risks of applicated SCM systems. They deliver a well elaborated explanation why distributed SCM has a higher learning curve. This finding is a important principle for dealing with SCM.

SCM systems are designed to handle plain text files, like those used for source code. After a file has undergone configuration management and had an initial transfer into the repository, the system stores only a delta of the changes for every new transaction. With this requirement the repository is more efficient and needs less disk storage. This implies binary files like office documents should not be stored in SCM repositories because the system cannot calculate a delta and will always store a complete new copy of the file, if it has been changed. A solution for dealing with binaries, like dependencies or third party libraries, are Repository Managers which were introduced in section II.

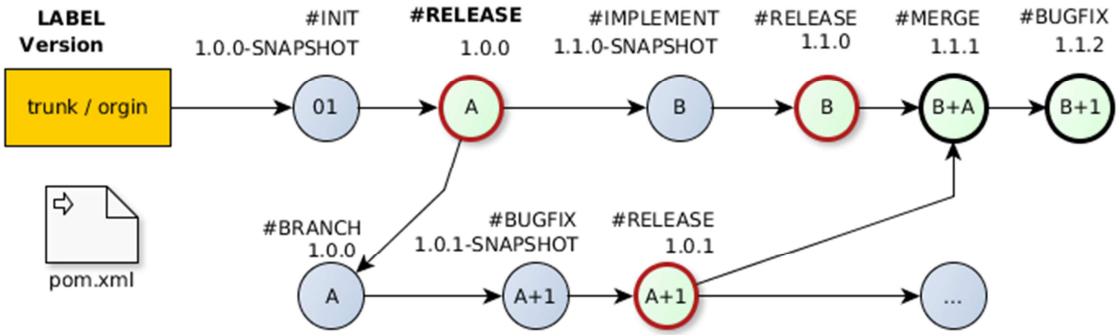

Figure 2. *Changes in the POM, based on Semantic Versioning.*

At this point some performance issues for SCM have to be taken in consideration. This is of outstanding importance, because it defines how a repository should be organized. Large projects with a code repository up to 1 GB take a long time for a checkout, even though there is only a small subset of files that are chosen. 20 minutes and more are very common. The reason for this effect is the size of the repository itself. When it contains a lot of files it takes more time to calculate the internal tree. The best solution for a high performance repository is: Only text files and just one independent project or module per repository.

In continuation surges question how files are represented in a SCM. As an example we remember the small Java library project with the Maven build logic. The build logic is represented as an XML file and contains the entry <version>. This entry defines the version number of the artifact and starts with an initialization of 1.0.0-SNAPSHOT. The procedure to increase the version number strictly follows the Semantic Versioning. Figure 2 visualizes several steps between two releases. For each revision a label describes the process and the version number show the value in the POM file. This graphic is an extension with a detailed view of figure 1.

In reality things are never like explained in theory. Initial assumption often create a big dilemma in automation processes when it comes to execution. It is very easy to claim, that in a repository, the entry for version in the POM for releases is unique. For example, it means that there should not exist two revisions with a released version 1.0. But where humans work, mistakes will happen. For this reason we have the option to create tags into the SCM. Every revision in the SCM which represents a deployed release, will be tagged with the correct version number. Deployed releases are defined by a successful transfer of the binary artifact into the Repository Manager for collaborative usage.

4. Scenarios on Real Problems

We should focus our activities on special points in respect to the evolution of software projects. It is not useful to pay attention on each single revision. Let us highlight the Points of Interest (POI) and why they are special. In real projects with collaborative teams, it is quite common that a developer breaks the current build. The good news are: when Continuous Integration (CI) is applied in the process, these kind of problems will be detected very quickly and can be solved at the instance of them appearing [16]. But how a developer is able to break a build? This occurs when the changes get committed into the repository and some files are not included in the commit. A repair can easily and fast be done by adding a new commit with the missing files needed. In this case it is very important to realize that only the one who delivered an incomplete package is able to add the missing parts. Problems arise when this happens on a Friday evening and the person responsible is leaving the office for vacations the next two or tree weeks without checking that everything is in order, causing unnecessary pain in the continuation of the project. These things happen much more often than anyone would expect.

Another effect is called fast shots. These small and often repeated commits typically change only a few lines in just

one or two files. This happens when a user for some reason is not able to test his code or settings locally on his own machine. A simple scenario could be the manipulation of the CI Server build output without direct access.

A work flow for developers is the usage of particular commits in order to preserve intermediate steps of the work and allow an easy rollback. This procedure is only applicable in distributed systems or in environments without collaboration. The effect is quit similar. It will produce many revisions inside the SCM, which could get summarized to a single revision.

The Continuous Delivery approach for modern Web Applications is a quite different method compared to the classical release process [14]. This technique requires special strategies like the Feature Toggle Pattern [22] and a highly automated deploy pipeline. Also the usage of the SCM system is very advanced. Each feature is developed in its own branch and the Configuration- or Build Manager creates for each deployment a proper Integration Branch. The biggest challenge in this methodology are fast responses towards urgent problems arising. In the worst case it could be necessary to push out very quickly a new deployment with a full or partial rollback. During deployments database changes are very critical. This aspect could be discussed in a further paper. Databases are not implicitly part of the SCM, but there also exist techniques [23] to keep them under configuration management.

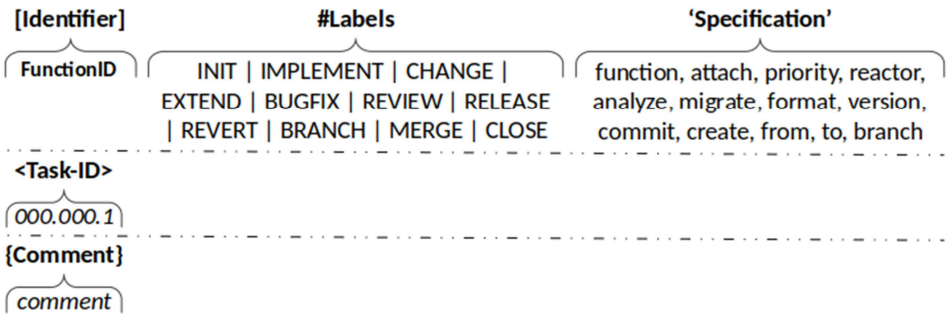

Figure 3. *Structure of a commit naming.*

As mentioned before, a release R inside an SCM is defined by several commits to the SCM. These commits are identified by the revision r. The lowest amount of revisions between two release is one, but there is no limit concerning to the upper boundary. Special Points of Interests inside an SCM are released revisions which can formally defined by (2).

$$R = \{r\,1, r\,2, r\,3, r\,n+1, \ldots, r\,x\} \quad (1)$$

$$POI: = \Delta\,\text{Release}\,(R; R + 1) \quad (2)$$

By this interpretation we are able to develop metrics which show a real project growth and do not just produce an output [13]. The paper of P. Kaur and H. Singh contains a collection of metrics related to their VVCT SCM [9]. An adapted suggestion for possibilities to compare project evolution is:

1) the amount of BugFix releases in a minor branch,
2) an count of revisions between two release,
3) the growth between minor and major release (e.g. Line of Codes),
4) a direct comparison between the current trunk and a previous release,
5) two selected releases,
6) a comparison of an release R and its replacement.

For example the amount of BugFix releases for a minor release allows a conclusion about the quality situation of a project. It is very important to understand the reasons to improve program stability and reduce the number of BugFixes. A classification for changes is described by Swanson [1]. An overview of the project based on these classifications of BugFixes should detect the issues that have to be changed to accomplish high quality.

5. A Vocabulary for SCM Commit Messages

In the early times SCM systems were used for synchronizing source code between developers. Typically users were not paying too much attention to write well formulated explanations about their changes. In many instances they were not leaving any description about what they did. Another extreme was that comments like update build logic frequently appeared in the history. An explanation of everything and nothing without saying what was changed or why. It could either be a version update of an existing library or the addition of a new dependency leading to a heavy time-consuming work in order to identify the points of interest in the commit history. Manual checks between the version with a Diff Tool would be necessary to locate the Line of Code that may have to be changed again. Guidelines have been introduced on how to write a well formulated commit message to solve this problems. A short selection of these guides published on the internet: [24-26] It was discovered by companies that the approach to apply well formulated descriptions of SCM revisions can improve productivity in teams. By exploring new projects on Source Code Hosting Services like GitHub or Sourceforge the quality of commit messages was increasing in the last years.

Based on these recommendations and the experience gained as of today, a vocabulary should be introduced for

writing easier and more efficient commit messages. This simple-to-use standardization could help to visualize the evolution of a project more clearly. By very precise and short explanation of every revision readers do not get flooded with information. This allows analysts to see patterns of process leaks more quickly and increases the team productivity. The usage of a defined structure also allows an automatism to parse the commit messages. The result can generate programmatic presentations of diagrams readable by humans. Naturally this approach is not only limited to SCM. Another usage could be for communication in meetings with strict time limitations, for example in the agile method Scrum.

The vocabulary for SCM Commit Messages follows a defined structure which is shown in figure 3. The composition contains a mandatory first line and includes a FunctionID, label and a short specification. The second and third line is optional and contains the TaskID from the Issue Management System and a description of the more detailed explanation. Our suggestion for the vocabulary covers most SCM work flows. It may will be that some companies need adoptions to implement this solution in their processes. For this reason the definition is flexible and allows extensions.

1) #INIT - the repository or a release.
   a) repro:documentation / configuration...
   b) archetype:jar / war / ear / pom / zip...
   c) version:<version>
2) #IMPLEMENT - a functionality.
   function:<clazz>
3) #CHANGE - a functionality.
   function:<clazz>
4) #EXTEND - a functionality.
   a) function:<clazz>
   b) attach:<clazz>
5) #BUGFIX - a functionality.
   priority:critical / medium / low / design
6) #REVIEW - an implementation.
   a) refactor:<function>
   b) analyze:<quality>
   c) migrate:<function>
   d) format:<source>
7) #RELEASE - an artifact.
   version:<version>
8) #REVERT - a commit.
   commit:<id>
9) #BRANCH - create.
   a) create:<name>
   b) stash:<branch>
10) #MERGE - from another branch.
   a) from:<branch>
   b) to:<branch>
11) #CLOSE - a branch.
   branch:<name>

As first entry a FunctionID is recommended and not the TaskID of the Issue Management. This decision is based on the experience that functionality could spread in different tasks. In longtime projects it could happen that for some reason the Issue Management System needs to be replaced by another one. Not all projects are connected to Issue Management, especially when they are small or just a prototype. These circumstances proved to be decisive to define the TaskId as optional and move it to the second line. With a FunctionID it is easier to identify parts that should be linked. Sometimes there exist transfers into the repository that cannot be assigned to a dedicated function. These commits are often related to activities of the Build- and Configuration Manager. As best practice an ID should be established which corresponds to these activities. Some examples related to the defined labels are:

1) [CM-00] INIT;
2) [CM-10] REVIEW;
3) [CM-20] BRANCH;
4) [CM-30] MERGE;
5) [CM-40] RELEASE;
6) [CM-50] build management.

The mightiness of this approach is its simplicity and how it can be included in existing projects. The rule set does not contain any additional complexity and the process is quite easy to understand. A short example will demonstrate the usage and a full example is provided in section VI. A change in the POM file to update the version of the test framework could be commented as follows:

[CM-50] #CHANGE ‘function:pom’
<QS-23231>
{Change version number of the dependency JUnit from 4 to 5.0.2}

6. Release Process

The sample project in section II is not only fictive. The Together Platform (TP) available on GitHub [26] was initiated to study techniques on real conditions. Hence Git is the SCM tool of the choice. As client SmartGit is recommended because of platform independence and it offers plentiful advanced functionality.

For better comprehension of our approach of writing commit expressions we use the TP-CORE project, from initialization of the repository to its first release. No TaskIDs for the revisions exist due to the project not being connected to an Issue Management System. We use an excerpt of TP-CORE to demonstrate the approach because between the initial commit and the first published release 1.0.2 exist over 70 revisions in the repository. The project also contains a set of 12 functions which do not need to be included completely in our sample. Only three functions were selected for demonstration:

1) CORE-01 Logger;
2) CORE-02 genericDAO;
3) CORE-05 ApplicationConfiguration.

This cuts the revisions in half and shows enough complexity avoiding readers falling asleep.

The condition for a first release was the implementation of all 12 functionalities. The overall test coverage has reached more than 85%. Code smells detected with checks by

Findbugs, Checkstyle, PMD et cetera have been removed. For an facilitate explanation, we add a revision number before the FunctionID. TP-CORE Commit Messages:

01[CM-00] #INIT 'archtype:jar'
{Initial the repository for Java JAR library.}
02[CORE-01] #IMPLEMENT 'function:Logger'
{Application wide standard logger.}
03[CORE-02] #IMPLEMENT
{Generic Data Access Object Pattern for centralized database access.}
04[CORE-05] #IMPLEMENT 'function:AppConfigDO'
{Domain Object for application configuration.}
05[CM-10] #REVIEW 'analyze:quality'
{Formatting, fix Checkstyle hints, JavaDoc & test coverage}
06[CORE-05] #IMPLEMENT 'function:ConfigurationDAO'
{Add the ConfigurationDAO implementation.}
07[CORE-05] #EXTEND 'attach:tests'
{Create test cases for Bean Validation.}
08[CORE-01] #EXTEND 'function:Logger'
{Add new Method to detect the configured LogLevel.}
09[CORE-05] #EXTEND 'function:AppConfigDO'
{Change Primary Key to UUID and extend tests.}
10[CORE-05] #CHANGE 'function:AppConfigDO'
{Rename to ConfigurationDO and define table indexes.}
11[CORE-02] #EXTEND 'function:GenericDAO'
{Add flushTable, countEnties and optimize.}
12[CORE-05] #EXTEND 'attach:tests'
{Update test cases for application configuration.}
13[CORE-05] #EXTEND 'function:ConfigurationDAO'
{Update the implementation for ConfigurationDAOImpl.}
14[CORE-01] #EXTEND 'function:Logger'
{Add method for exception handling.}
15[CORE-05] #EXTEND 'function:ConfigurationDO'
{Add field mandatory.}
16[CM-10] #REVIEW 'migrate:JUnit'
{Migrate Test cases from JUnit4 to JUnit5.}
17[CM-10] #REVIEW 'analyze:quality'
{Fix JavaDoc, Checkstyle & Findbugs.}
18[CM-50] #EXTEND 'function:POM'
{Update SCM connection to GitHub.}
19[CM-50] #EXTEND 'attach:APIguards'
{Attach annotation for API documentation.}
20[CORE-05] #REVIEW 'refactor:ConfigurationDO'
{FindBugs: optimize constructor parameters.}
21[CORE-02] #BUGFIX 'priority:design'
{Fix FindBugs hint: visible modifier.}
22[CM-50] #EXTEND 'attach:site'
{Extend MVN site configuration.}
23[CORE-02] #BUGFIX 'priority:high'
{Fix spring DAO configuration.}
24[CORE-05] #IMPLEMENT 'function:ConfigurationService'
{Implement basic functionality for ConfigurationService.}
25[CM-10] #REVIEW 'analyze:quality'
{Remove all compiler warnings, FindBugs, Checkstyle & PMD Hits.}
26[CORE-05] #EXTEND 'attach:ConfigurationService'
{Add JGiven test scenarios.}
27[CM-40] #RELEASE 'version:1.0'
{Release artifact to version 1.0}
28[CM-40] #RELEASE 'version:1.0.1'
{Change POM GroupId to Maven Central conventions.}
29[CM-00] #INIT 'version:1.1'
{Start implementation of version 1.1.0.}
30[CM-50] #MERGE 'from:1.0.1'
{Integrate GAV POM changes to trunk.}
31[CM-40] #RELEASE 'version:1.0.2'
{Include PGP signing.}
32[CM-20] #CHANGE 'function:Constraints'
{Add Constraints.VERSION to 1.1}
33[CORE-01] #EXTEND 'function:Logger'
{Default loader for logback.xml configuration files in the application DIR.}

Considering the previous example, we see that a limitation to around 80 - 100 characters for the first line is recommendable. Displaying the history with any client could get very messy if the first line has no size restrictions. The log output of the commit messages does not display the branch and tag operation, a behavior of Git. These revisions do not appear in any history list by browsing GitHub. Revision 28 is a branch based on revision 27. The branch is named as 1.0. Releases are published in consonance with the convention to be labeled, revision 31 tagged as Release 1.0.2. The revisions 28 and 31 are part of branch 1.0.

In this constellation we are able to see an important detail for dealing with branches. A branch will only be created when it is necessary. Usually BugFix branches do not have their own build plans on CI Servers and are managed manually. The primary arguments for this practice are to reduce the administrative overhead for the CI Servers. Companies that orchestrate their applications by web services or modules loose capacities by binding their recourses in this kind of activities.

7. Conclusion

"There is nothing permanent except change." - Heraclitus

The whole infrastructure of commercial software projects contains a lot of independent fragments which share information over all development cycle. In projects we are overloaded by documentation production processes. The high amount of all this information inhibits profoundly comprehension and handling capabilities. Applications are getting more complex and bigger resulting in the necessity to establish more efficient ways to deal with information accumulation. There exists a giant overhead of managing documents like release notes, release plan, issue management, quality reports, statistics & metrics, documentation, architectural documents and BugFix lists.

Typically each tool stores its data in its own structure. This makes changes to other tools, that might fit better, risky and expensive.

Companies know the effect that developers feel uncomfortable having to track their work in Issue Management tools like JIRA resulting in them trying to hide their part of the work flow as much as possible. Tasks will be opened up when they are almost done or already finished. The information on how many project days were spent for a function covers more the expectations and less the reality with the intent that developers can escape a bit from the daily pressure of productivity. Often developers are forced to spend their time with data acquisition for management controlling instead of programming resulting in low cost efficiency of a project and even additional and unplanned costs. Developers dislike this kind of activities because it keeps them away from their actual work: development. This is what makes the simple approach towards human readable and machine processable commit messages attractive and more convenient. The most important fact is that no extra costs are generated applying this method to existing processes.

We are enabled to generate several reports based on real data if SCM repositories can be populated with additional information. Impact assessments could be more efficient and accurate when they are created by facts and not emotionally blended.

8. Future Work

The idea to make information inside SCM systems more transparent is not just limited to commit messages. Another obvious point for future research is the history command. In the paper of Abram Hindle and Daniel M. German a query language for source control is introduced [7]. The idea of SCM Language could be picked up and transformed applying it to a specific solution. This work would use the Domain Driven Development paradigm to model an own SCM language based on Domain Specific Language (DSL) concepts - leading to the discovery of real world DSL solutions allowing for quick construction of a viable prototype or application based upon certain specifications.

Also a point which boldly comes to mind after reading the paper of Fischer et al., is the inclusion of released information into SCM [4]. This approach should not fully be automated due to its requirement of an advanced knowledge about branching and merging. A small self written extension could be a probable solution. A short tutorial for Git suggests certain possibilities.

Acknowledgements

Special thanks to Joachim Reiter and Harald Kaufmann for spending their time to review this document. Their feedback was very productive.

References

[1] E. Burton Swanson, 1978, The Dimension of Maintenance.

[2] Walter F. Tichy, 1985, RCS - A System for Version Control.

[3] Chuck Walrad and Darrel Strom, 2002, The Importance of Branching Models in SCM.

[4] Michael Fischer, Martin Pinzger, Harald Gall, 2003, Populating a Release History Database from Version Control and Bug Tracking Systems.

[5] Filip Van Rysselberghe and Serge Demeyer, 2004, Mining Version Control Systems for FACs (Frequently Applied Changes).

[6] Louis Glassy, 2005, Using version control to observe student software development processes.

[7] Abram Hindle and Daniel M. German, 2005, SCQL: a formal model and a query language for source control.

[8] Yongchang Ren, Tao Xing, Qiang Quan, Ying Zhao, 2010, Software Configuration Management of Version Control Study Based on Baseline.

[9] Parminder Kaur and Hardeep Singh, 2011, A Model for Versioning Control Mechanism in Component- Based Systems

[10] Shaun Phillips, Jonathan Sillito, Rob Walker, 2011, Branching and merging: an investigation into current version control practices.

[11] Abdullah Uz Tansel and Ali Koc, 2011, A Survey of Version Control Systems.

[12] Christian Bird et al., 2014, Transition from Centralized to Decentralized Version Control Systems A Case Study on Reasons, Barriers, and Outcomes.

[13] Norman E. Fenton and Shari Lawrence Pfieeger, 1997, PWS Publishing Company, Software Metrics - A Rigorous and Practical Approach 2nd Edition, ISBN O·534·95425·1.

[14] Jez Humble and David Farley, 2010, Addison-Wesley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, ISBN 0-321-60191-2.

[15] Scott Chacon and Ben Straub, 2014, Apress, Pro Git 2nd Edition, ISBN 978-1-4842-0077-3.

[16] Mike Clark, 2004, The Pragmatic Bookshelf, Pragmatic Project Automation, ISBN 0-9745140-3-9.

[17] Dave Thomas and Andy Hunt, 2003, The Pragmatic Bookshelf, Pragmatic Version Control with CVS, ISBN 0-9745140-0-4.

[18] Mike Mason, 2010, The Pragmatic Bookshelf, Pragmatic Guide to Subversion, ISBN 1-934356-61-1.

[19] Sonatype Inc. (2017), Maven Central, https://search.maven.org

[20] Git (2022), Git Documentation https://git-scm.com/book/en/v2/Git-Branching-Branching-Workflows

[21] Vincent Driessen (2022), Git Flow, https://nvie.com/posts/a-successful-git-branching-model/

[22] Martin Folwer, (2022), Feature Toggles, https://www.martinfowler.com/articles/feature-toggles.html

[23] Red Gate Software Ltd. (!999), Database Versioning, https://flywaydb.org

[24] Cris Beams (2022), Writing Git Commit Messages, https://chris.beams.io/posts/git-commit/

[25] Who-T (2009), On Commit Messages, http://who-t.blogspot.mx/2009/12/on-commit-messages.html

[26] Microsoft (20), GitHub Open-Source repository, https://github.com/ElmarDott/TP-CORE/

Biography

Marco Schulz, also kown by his online identity Elmar Dott is an independent consultant in the field of large Web Application, generally based on the JavaEE environment. His main working field is Build-, Configuration- & Release-Management as well as software architecture. In addition his interests cover the full software development process and the discovery of possibilities to automate them as much as possible. Over the time of the last ten years he has authored a variety of technical articles for different publishers and speaks on various software development conferences. He is also the author of the book "Continuous Integration with Jenkins" published 2021 by Rheinwerk.