Computer Science • Vol. 8 Special Edition • 2007

Sławomir P. Maludziński*

SOFTWARE CONFIGURATION MANAGEMENT FOR MULTIPLE RELEASES: INFLUENCE ON DEVELOPMENT EFFORT

Software Configuration Management (SCM) evolves together with the discipline of software engineering. Teams working on software products become larger and are geographically distributed at multiple sites. Collaboration between such groups requires well evaluated SCM plans and strategies to easy cooperation and decrease software development cost by reducing time spent on SCM activities – branching and merging, that is effort utilized on creation of revisions ('serial' versions) and variants ('parallel' versions). This paper suggests that SCM practices should be combined with modular design and code refactoring to reduce cost related to maintenance of the same code line. Teams which produce several variants of the same code line at the same time should use approaches like components, modularization, or plug-ins over code alternations maintained on version branches. Findings described in this paper were taken by teams in charge of development of radio communication systems in Motorola GEMS divisions. Each team collaborating on similar projects used different SCM strategies to develop parts of this system.

Keywords: *software configuration management, scm, multisite, software engineering*

ZARZĄDZANIE KONFIGURACJĄ OPROGRAMOWANIA DLA WIELU WERSJI: WPŁYW NA KOSZT WYTWARZANIA

Zarządzanie konfiguracją oprogramowania (SCM) ewoluuje razem z dyscypliną inżynierii oprogramowania. Zespoły pracujące nad wytwarzaniem oprogramowania stają się coraz większe oraz niejednokrotnie fizycznie znajdują się w ośrodkach położonych na różnych kontynentach. Współpraca pomiędzy takimi zespołami powinna opierać się na dobrze przygotowanych planach zarządzania konfiguracją oprogramowania. Niniejszy artykuł sugeruje, że praktyki zarządzania konfiguracją oprogramowania powinny być połączone z fazami projektowania oraz refaktoringiem kodu, tak aby zmniejszyć koszt związany z utrzymaniem tej samej linii kodu. W opinii autora artykułu, każdy z wariantów oprogramowania powinien być wytwarzany przy użyciu technik innych niż związane z zarządzaniem konfiguracją. Zespoły pracujące nad kilkoma wariantami tej samej linii kodu powinny przedkładać biblioteki i komponenty nad modyfikacje dokonywane i utrzymywane na gałęziach wersji. Doświadczenia opisane w artykule zostały nabyte przez zespoły pracujące w dziale firmy Motorola zajmującym się wytwarzaniem oprogramowania dla systemów radiokomunikacyjnych.

Słowa kluczowe: *zarządzanie konfiguracją oprogramowania, inżynieria oprogramowania*

* email: slawomir.maludzinski@motorola.com

1. Introduction

Astro and Dimetra are respectively American and European variants of public safety communication system. Both use similar hardware and software for digital radio communication. Although they implement different communication standards, APCO and TETRA respectively, many aspects of their functionality are similar. Hardware and software for mobile infrastructure is developed by many teams in USA, Europe (Poland, Denmark) and other locations. In this paper we focus on SCM strategies used by Astro team in Schaumburg (SCH), IL, USA; Dimetra team in Krakow (KRK), Poland, and Dimetra team in Copenhagen (COP), Denmark. Statistical comparison with STM team which uses an approach similar to the proposed in this paper is given to verify finding of the author of this article.

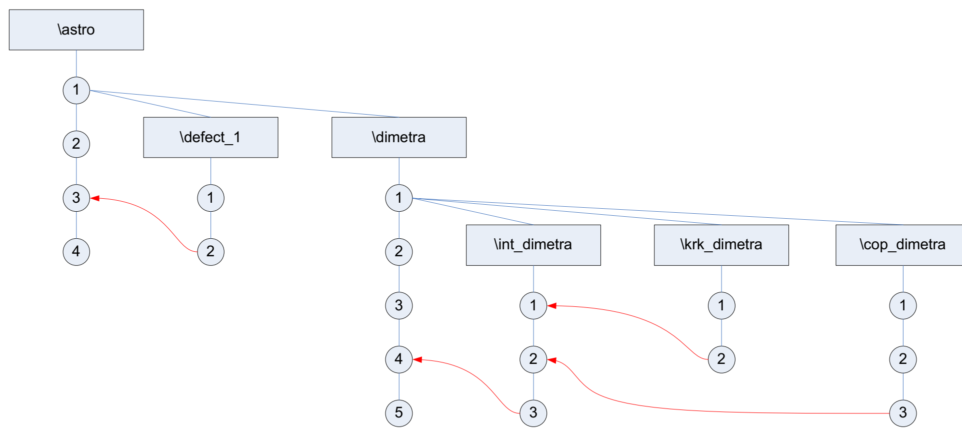

Fig. 1. Astro Dimetra code development branches

The aforementioned teams use similar SCM [4] plans to develop new features, resolve defects and release software. Figure 1 presents typical scenario where new features are developed on separate branches, and when tested and approved, back-merged to the main development branch. This SCM plan is used by SCH Astro team. A little more complicated SCM plan is used by KRK and COP Dimetra teams. An integration branch is created to integrate changes made at these sites. Additional branch is used as Clear Case imposes development on a branch which pertains to the same physical site i.e. software engineer must create a separate branch in their location to make changes. This however, is not relevant to the discussion presented in this article. After the work is completed, development teams in KRK or COP merge differences created on development branches to integration branch. This reduces time spent by release team in charge of system builds. Release team will not have to resolve any potential conflicts; merge to the release branch should be trivial.

Collaboration between Astro and Dimetra is quite interesting in terms of variants development of network management (NM) component used to configure radio net-

works. Astro develops major parts of NM software which is then merged by Dimetra. These are mostly newer features, smaller changes as well as defect fixes. SCH Astro team changes are very distinct. Their integration with Dimetra branches happens less often – every half a year on the average. KRK and COP teams, on the other hand, co-operate very closely. Their branches are constantly integrated and any modifications are immediately visible and used by teams at both locations.

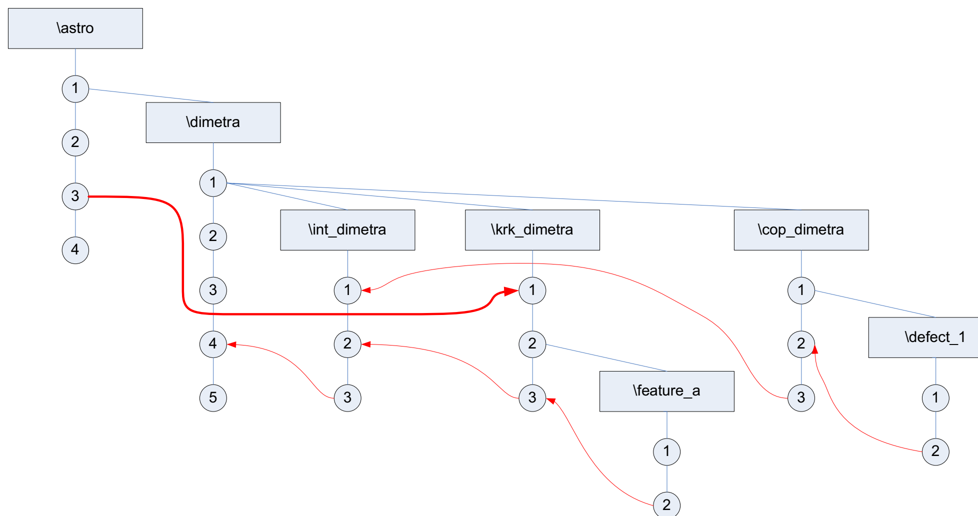

Fig. 2. Dimetra separate development with infrequent merges from Astro

A typical example of such collaboration is shown in Figure 2. Astro changes are merged from the \`main` branch to ...\`dimetra` branches. Krakow and Copenhagen work on their development branches to deliver functionality which is Dimetra specific. An attentive reader will notice that some changes are taken from Astro to Dimetra. Every change made by Astro team needs to be incorporated properly, built and tested. Less frequently changes made by Dimetra are "back-merged" to Astro. This causes that in practice software is developed on several branches. Astro differences between revisions are constantly selected, copied and pasted into Dimetra's line of code. The burden of subsequent operations makes large amount of tasks to be repeated by both teams. Although in some cases inevitable, such behavior contributes to significant loss of productivity and elongated development cycle.

To ensure that fixes are properly merged is not a trivial task and also is a very expensive one. Changes to the other project need to be made manually as many merge conflicts occur. Next such modification is to be tested and verified which also consumes human effort.

Similar effects occur in less complicated situations. Figure 3 outlines a situation when a defect was fixed by KRK Dimetra team. Until it is merged by Astro, Dimetra will have to deliver subsequent revisions of the file on their own branch ...

`\dimetra`. If Astro makes modification it will have to be merged to Dimetra. The process of merge, back-merge, build and test will continue on separate code development branches.

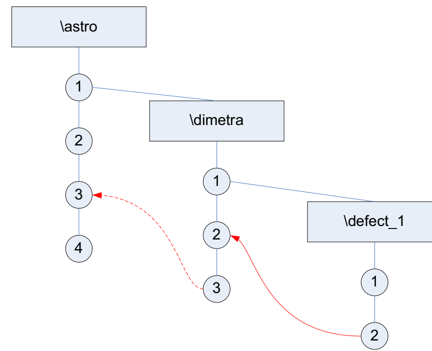

Fig. 3. Defect fixed only at NM Dimetra

2. Branch-merge operations

Article [1] discusses two branching strategies – branch-by-release and branch-by-purpose. Branch-by-purpose uses an approach where additional branch is created for tests and builds. Development remains on the main trunk. It has several advantages over the more "traditional" approach where branches are created for every subsequent release. For example, development may continue on the main branch which does not confuse developers and allows system builders prepare next version for testing. Defect fixes are easily picked up and can be selected for emergency releases. Authors of the paper also suggest creating additional branches to develop newer features or to correct defects. This SCM strategy resembles the one used by Astro and Dimetra. However, after code chill out and freeze – when no newer features are introduced and code baseline has stabilized – parallel development branches are integrated to the main branch.

This paper emphasizes that integration to the main line of development is essential to reduce effort spent on software development in subsequent releases. Obviously such operation in many cases is not straightforward as it requires code refactoring and unit tests or even regression tests to be performed. Modules, plug-ins and libraries can be helpful to select invariants and differences between releases. Next, such functionality can be developed separately without an additional effort spent on branching and merging. Of course architecture and refactoring needs to be performed by all teams engaged in product development otherwise some unresolved conflicts may happen.

No matter whether variants of code are developed in closely related projects or at loosely coupled versions, which often have separate requirements, teams should try to integrate their work. Although releases start to differ over time, teams developing

them should merge and select common parts, also identify differences and move them to newer elements (files) which will be maintained on their own. This will later on lead to saving time spent on the cross release merges and verification. Such code is easily found by examination of version trees. Frequent long merges and back merges clearly show that such code should be revised and altered to avoid continuous merging in the future.

Code refactoring needed to bring the variants together will select code specific to each release. Starting from this point in time – no or only small amendments will be needed in the common part. Software modules will be developed separately – avoiding merges and expensive verifications. After the system is deployed, in an ideal scenario appropriate code will be customized – that is how version tree structure is helpful to identify code which is problematic to maintain and causes a lot of burden and hassle.

Merges between releases (variants of code) resemble code copying in many aspects. When not back integrated same code needs to be maintained in many copies. This stands in contradiction to well known good software engineering practice to avoid code copying.

3. Case study

Similar findings were observed by STM team which develops code for Astro and Dimetra releases of another network device. Quantitative analysis of code unveiled that only 3 to 4 per cent of code differs between releases. Engineers were also anxious about work consuming task related to merges of features and defects between different releases. Also, after the merge operation resulting variants of code were very similar. This indicated that something can be done to improve efficiency and lessen development effort.

Teams decided to change their strategy and keep only one line of code for both releases. This decision was taken to simplify code maintenance by the cost of its development. Parts of software which were specific to each of releases were executed conditionally. In the very first versions of their system only two versions had to be maintained – Astro and Dimetra.

```
if ( Astro ) ...
if ( Dimetra ) ...
```

Soon, as more releases appeared, they needed to be maintained as well. More conditionals had to be introduced.

```
if ( Astro4.x ) ...
if ( Astro5.x ) ...
if ( Dimetra ) ...
```

Things got even worse after several more releases – major and minor version of the system was distinguished and based on this information appropriate action taken.

```
if (( major_ver(Astro) == 6 &&
          minor_ver(Astro) <= 5 ) ||
          major_ver(Astro < 6 ) ...
if ( Dimetra ) ...
```

Of course such approach was not convenient any longer. The watershed in code maintenance was the decision to create a library which related features to releases. The library was customized by a configuration file and gave answer to a simple question if a feature belongs to a given release.

	Astro4.x	Astro5.x	Dimetra
FeatureX	Yes	Yes	Yes
FeatureY		Yes	

```
if ( featureIsOn(FeatureX) ) ...
```

Introduction of guards [Ref. 3] allowed simplifying statements which decided whether to enable given functionality. General code, not specific to every release had to be written outside of conditional variables. Features which were added to specific releases were surrounded by conditional statements. Starting from this point in time code was executed based on run-time information. It is worth noticing that no significant reduction in code performance was noticed due to insertion of these statements.

The obvious drawback of the solution is that conditionals are placed in the code. In some cases, for example when using binaries or third party products, conditional statements are not possible. Configuration files, specific to each release in many cases cannot be customized. This, however often is not a problem. Prevailing number of merges of those files is a trivial thanks to their simple syntax. We argue that this inconvenience is worth paying – it seems that positives are prevailing. Whenever a defect is fixed it is corrected for every release. Theoretically there is no time spent on introduction of features to newer releases. Merely appropriate entries in the release/feature matrix need to be set. Developers need not perform mundane work of merges between lines of code and configuration management becomes much easier – code is being maintained on fewer numbers of branches.

4. Statistical information

Team members, who work on newer features and resolution of defects, report time spent on their activities on a weekly basis. This information is gathered in a special tool which lets track progress of project activities. Obviously as this data is entered by engineers manually it is not very precise. However we one may analyze these reports to get a general feedback about state of a given project.

The Table 1 presents data about defects which were resolved by different teams. Merges are resolutions of same defects which were ported to other releases which are begin developed at the same moment.

Table 1
Effort spent to resolve a defect and merges to different releases

	effort / (defect or merge total) [h]	#defects + #merges	effort / defect [h]	#defects	effort / merge [h]	# merges
NM Astro	14.03	550	–	478	–	72
NM Dimetra	20.02	244 + 128*	21.43	204 + 77*	12.8	40+51*
STM Astro & Dimetra	14.60	184	23.75	110	0.99	74
*not taken into account						

At the time of writing this paper there is no statistical information available which would distinct Astro defects and merges. Quite interesting is the comparison of effort spent on merges between NM Dimetra and STM Astro & Dimetra. Statistics unveil over 10 time's larger difference between engineers who merge resolutions at these teams. Certainly this value should be a bit lower as it takes longer to resolve a defect at NM Dimetra; however a significant amount of effort is saved at STM Astro & Dimetra. Further factors may also impact this figure, but obviously our findings have been confirmed in practice.

5. Further work

Quality assurance teams gather data which helps to asses projects based on diverse types of metrics. Work effort to develop a feature, resolve defect or defect's cycle times are measured and can be compared between projects. Such analysis would give us supplementary information about savings which can be gained using the proposed methodology. Such data would be helpful to additionally verify arguments of this paper. Especially, statistics on development of larger units of code, being created by several software engineers would be useful. Gathering of such information is not a trivial task, though. Small number of features is likely not giving enough material which can be analyzed statistically. Furthermore, it is not easy to make simple comparisons between projects as many factors have influence on features' development time and effort. Amidst most considerable are variations in software processes, laboratory resources availability or review and test procedures. Detailed study would have to be made to compare such results.

Giving hints how to refractor code based on version tree structure is very complicated and seems ambiguous. Every situation is different and requires specific analysis. A lot of depends on code structure, differences, and common parts. Complicated merges and complex version trees point that code is worth considering to be changed.

One of future extensions would be survey of most appropriate techniques which can be utilized in different scenarios.

6. Conclusion

SCM plans used by teams working together at multiple sites are usually very long and detailed. Variants of code are created and maintained following those guidelines. Different releases (variants) are created by modifications between branches of code. Although this is the most straight-forward technique, it should be used in as few cases as possible. Variants of software should rather be delivered by other means than maintenance on version branches. In the long term this approach will cause an emergence of libraries, components and forms of code which easy customizations. Findings presented in this paper show that this approach is undoubtedly worth considering.

References

[1] Walrad C., Strom D.: *The Importance of Branching Models in SCM*, IEEE Computer, 35(9), 2002, p. 31–38

[2] Sommerville I.: *Software Engineering*, AddisonWesley, Reading, MA, 6th Edition, 2001

[3] Fowler M.: *Refactoring: Improving the Design of Existing Programs*, Addison-Wesley, 1999

[4] Conradi R., Westfechtel B.: *Version Models for Software Configuration Management*, ACM Computing Surveys, 30(2), June 1998, p. 232–282

[5] Allen L., Fernandez G., Kane K., Leblang D., Minard D., Posner J.: *ClearCase MultiSite: Supporting geographically-distributed software development.* In Jacky Estublier, editor, Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops, number 1005 in Lecture Notes in Computer Science, Springer Verlag, October 1995, p. 194–214