Towards a model of versioning domain (English)

Vladimir Jotov
Veliko Tarnovo University
"St. Cyril and St. Methodius"
email: vjotov@acm.org

Abstract (English)

Improvement in software development process is a permanent challenge in software engineering where version control systems are permanently supporting the process. For achieving follow improvement we need a better understanding of the versioning domain.

Starting from granularity elaboration of versioned object in this article we shall try to actuate toward the issue of classification of versioned object according to their place in software development process. Moreover, the article is to be a trigger for augmentation the scope of versioning systems by supporting of trace links among versioned objects.

***Keywords**: Versioning; versioned object granularity; version traceability.*

К модели управления версиями (Russian)

Владимир Йотов
Велико Тырновский Университет
"Св. Кирил и Св. Мефодий"
email: vjotov@acm.org

Abstract (Russian)

Совершенствование процесса разработки программного обеспечения является постоянной задачей в области инженерии программного обеспечения, где система контроля версий является системой жизнеобеспечения процесса. Чтобы достичь последующих улучшений, нам необходимо лучше понимать предметную область версий.

Начиная с детализаций версионизованных обьектов в этой статье, мы постараемся обсудить вопрос классификации версионизированных обьектов в зависимости от их места в процессе разработки программного обеспечения. Кроме того, цель статьи - стать толчком к расширению сферы систем версионизирования, включая прослеживающие связи среди версионизированных обьектов.

***Keywords**: Версионизация, грануляция версионизырованных обьектов, проследимость версии.*

1. Introduction

In [3] the author presents a roadmap of the main challenges in front of SCM systems. And as one of the pointed areas for improvement is versioning data management and workspace support from semantic perspective.

Before that, we would like to present a three layer model for Configuration Management (CM) [1]:

- SCM Primitives supports functionality for primitive structure – basic versioning, object locking, access control;
- SCM Protocol supports implementation process – data check in/out, workspaces, transactions management, and coordination of change set.
- SCM Policy supports organization specific procedures – change request handling, Quality Assurance (QA) procedures and etc.

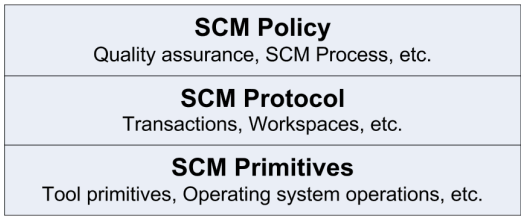

Figure 1 Federated SCM architecture proposed by Brown [1]

In contrast to the federated SCM architecture in this article we shall try to present a viewpoint for achieving improvement in the above described area based on domain data model. Therefore, after presenting some general principles in versioning area, the main goal of the article is to elaborate the granularity and composition of the versioned objects, as this has impact over the software development process and change management process.

The next section is an introduction in the versioning domain. Section 3 presents the natural evolution of versioning systems towards the configuration management systems. In that section we have tried to elaborate the configuration entity by decomposing versioned objects. Based on this and on the presented Section 4 version traceability you can find and in section 5 is presented the model of versioning domain.

2. Model of versioned object

Within the domain of versioning we are talking of versioned objects. The core characteristic of a versioned object is that it allows to obtain all statuses caused by its changes in time. On the other hand, a not-versioned object is an object that does not have this characteristic so we can get only its state after the last change. Any modifications over a not-versioned object overwrite its state [10]. In the presented versioning model we shall assume that the versioned object is a set of immutable not versioned objects that represents all statuses of the versioned object. Further in this article we shall refer to these immutable not-versioned objects with the term versioned primitives.

In [2, 8], the authors are using so the called version graph – directed acyclic graph where each node presents a specific version or versioned primitive of a versioned object. For a specific node we can have many outgoing arc to other nodes – this represents splitting of version. A node can also have several incoming arcs – presenting merging of several versions into one version of the object. Version graph allows us to trace the history of the objects so we could specify that the second core part of the versioned object is the version graph.

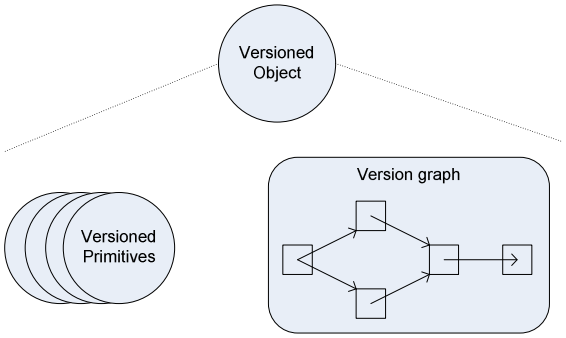

Figure 2 – Model of versioned object

3. Composition model of versioned objects

Systems for version control historically evaluate from checkout/check-in model of versioning to composition model [4]. The focus of the latter one is to manage the history of configuration. According to [3] the configuration in most of the versioning systems is not defined as an object, but as a query for extracting all needed components for the system. Hence, in the model presented we could regard the configuration as on a versioned object where its versioned primitives consist of *system model* and *version selection rules* [4]. Software system could be decomposed on several packages or modules. From development perspective the modules could be decomposed on classes and resources. The reverse scheme of decompositions – the rules for composition of these elements is an example for *system model* of the configuration.

In most versioning systems, files are the finest size of granularity for the versioned object [5, 6]. In [2] the author presents the paradigm of granularity of versioned objects. Here we will try to elaborate this paradigm starting from classes as main software construction element.

For the class we could determine that it has external and internal structure where the external structure specifies the class interface for interaction with outer world, what other classes have influence and are required for its normal behavior. The internal structure specifies the class internal behavior and interactions. As base class elements we could point its fields and methods.

In the class decomposition we shall use the term *meta-attribute* in order to avoid confusion of terms and misunderstandings. The meta-attribute is a piece of data that describes the class itself.

When we try to determine a class as unique functional part of the system we use its name and package, therefore these are the most important meta-attributes. There are different *type of classes* – *interface*, *abstract*, *normal class*, *final class*, *inner class*. And for inner class we have the meta-attribute *outer class*. According to the object-oriented

paradigm, a class could inherit another class. Some languages allow the inheriting of more than one class [12]. On the other hand, there are languages like Java [11] where the polymorphism is realized using interface. Therefore we can define the following two class meta-attributes – *list of inheritances* and *list of interfaces*. Usually in each class we use objects from other classes and all object-oriented languages provide mechanism for using data structures, defined in other external classes. This brings us to the next class meta-attribute – *list of imported classes*.

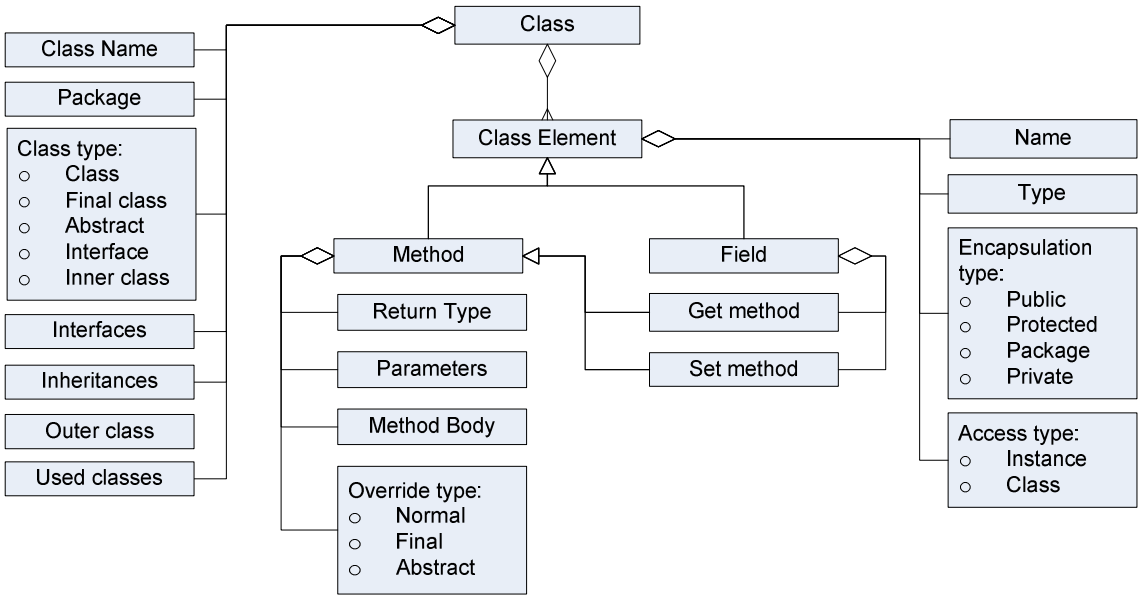

Figure 3 Model of class entity's elements and their composition

For fields and methods of a class, we can emphasize the following common meta-attributes: *name*, *type*, *visibility handler* and *access type*. Access type determines whether the level for accessing the element and the possible levels are via instance of the class or via the class itself. The visibility handler describes whether the concrete element is visible in outer classes or not. The most common visibility types in object-oriented languages are *public*, *protected*, *private* and *package*.

Method entity having the following meta-attributes extends the class element entity – *return type*, *list of parameters*, *method body* and *override type*. Using the override type we could specify whether the method is normal, final (i.e. it cannot be overridden) or abstract (i.e. the method has to be overridden).

In new object-oriented languages, like C#, we observe evolution of field entity. What is new for them is that they use special service methods known as setters and getters that allow defining ad-hoc control on accessing the field data in read and write access attempts. The fact that these methods share common interface for interaction gives us the idea of regarding them as part of the field entity. For example getters don't have parameters and it is always returning value of respective field type and the setters receive as a parameter value of the respective field type and they don't return any value.

On figure 3, the model of class entity's elements and their composition is presented.

4. Version traceability

The goal for software products is to satisfy some business needs described in requirements specification. Change in the requirements and scope of the project is commonly seen in the practice and this pushes the industry to use techniques and practices for managing them. Requirements traceability [7] is a good example of this. As the change is the nature of versioning and the possible impact on system components on change of the requirements this brings us the idea of using the version control for all artifacts in software development process.

Change in the requirements is only one of the *initiator* for launching of a change in the software. The other *initiator* is the reported system malfunctions from quality assurance and from software support phase. The *effector* is the system component(s) that is changed in order to satisfy the requirements from the *initiator*.

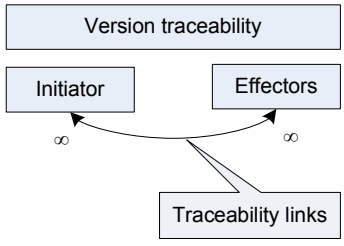

Figure 4 Version traceability

It is important to have clear vision over the changes in requirements, the impact of those changes on the end product. This leads us to the need of next level in the versioning systems – version traceability. Version traceability could be regarded as meta-data layer of the changes where it is described change in witch element forces change in the other elements. This layer allows us to measure the actual work effort required for achieving the required system behavior caused by each initiator. On the figure 3 we present the version traceability.

5. Model of versioning domain

As a summary of this article we would like to bring together all pieces and to have a complete model in the versioning domain. We are obligated to the versioned object model described in Section 2 as the fundamental in the domain. The fine granulation of versioned objects allows us to apply less complicated and more specialized algorithms for version merging but this will force applying more sophisticated algorithms for construction the software build.

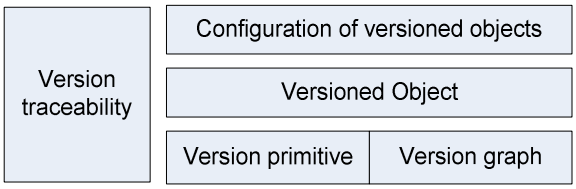

Figure 5 Model of versioning domain

The version traceability allows to evaluate the performed work and to receive the real price of the product. The proposed versioned traceability model pretends for contribution also in this area providing facilities for estimating effort of the requested change. For example in [9] the authors make attempts for estimate the effort and the price of particular change in the system based on tracing similar changes over the system in the past.

6. References

[1] Brown, A., S. Dart, P. Feiler, K. Wallnau, The state of automated configuration management. Tech. Rep. CMU/SEI-ATR-91, Software Engineering Inst., Carnegie Mellon Univ., Pittsburgh, 1991.

[2] Conradi, R., B. Westfechtel, Version models for software configuration management. ACM Comput. Surv. 30, 2, New York, 1998, pp. 232-282.

[3] Estublier, J., Software configuration management: a roadmap. In Proceedings of the Conference on the Future of Software Engineering (Limerick, Ireland, June 04 - 11, 2000). ICSE '00. ACM Press, New York, 2000, pp. 279-289.

[4] Feiler, P. H., Configuration Management Models in Commercial Environments, Pittsburgh, 1991

[5] Nguyen, T. N., E. V.Munson, J. T. Boyland, C. Thao, An infrastructure for development of object-oriented, multi-level configuration management services. In Proceedings of the 27th international Conference on Software Engineering (St. Louis, MO, USA, May 15 - 21, 2005). ICSE '05. ACM, New York, 2005, pp. 215-224.

[6]Nguyen, T. N., E. V. Munson, and J. T. Boyland, Object-oriented, structural software configuration management. In Companion To the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (Vancouver, BC, CANADA, October 24 - 28, 2004). OOPSLA '04. ACM, New York, 2004, pp. 35-36.

[7] Jarke, M. Requirements tracing. Commun. ACM 41, 12, 1998, pp. 32-36.

[8] Slein, J. A., F. Vitali, E. J. Whitehead, and D. G. Durand, Requirements for distributed authoring and versioning on the World Wide Web. StandardView 5, 1, Mar. 1997, pp. 17-24.

[9] Weiss, C., R. Premraj, T. Zimmermann, and Zeller, A. How Long Will It Take to Fix This Bug?, In Proceedings of the Fourth international Workshop on Mining Software Repositories (May 20 - 26, 2007). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 1, 2007.

[10] Whitehead, E. J. Design spaces for link and structure versioning. In Proceedings of the Twelfth ACM Conference on Hypertext and Hypermedia (Århus, none, Denmark, August 14 - 18, 2001). HYPERTEXT '01. ACM Press, New York, 2001, pp. 195-204.

[11] Брус Е., Да мислим на JAVA, том 1, SoftPress, Sofia, 2001, pp. 591

[12] Фейсон Т., Borland C++ Обектно-ориентирано програмиране - Част I, Nisoft, Sofia, 1994, pp. 400