Diploma Thesis

January 18, 2006

Developing a Meta Model for Release History Systems

Dane Marjanovic
of Bihac, Serbia and Montenegro (01 730 340)

supervised by
Harald Gall
Martin Pinzger

University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

Diploma Thesis

Developing a Meta Model for Release History Systems

Dane Marjanovic

University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

Diploma Thesis

Author: Dane Marjanovic, dane.marjanovic@gmail.com

Project period: 19.07.2005 - 19.01.2006

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgments

Many thanks to all people, that contributed to this thesis. Martin Pinzger, for the supervision and idea support. Michael Wrsch and Andreas Jetter, for providing the CVS implementation and tips to Hibernate. Also thanks to my fellow graduands for the proof reading of the text and useful ideas.

Abstract

The goal of this thesis is to construct a meta model for release history systems, based on SVN (Subversion)[1] and CVS[2]. The meta model will encompass the core concepts of versioning systems as they are present in the mentioned tools. The release history aspect will then be extended by an issue tracking data model for which we take the Bugzilla[3] data representation. With the meta model's semantics, one will be able to model random release history systems similar to CVS or SVN. Further the meta model will be able to model the release history aspect of CMS (Configuration Management Systems) such as ClearCase[4] or Visual Source Safe[5], as we will validate the meta model with the Rational ClearCase data model. The focus of this thesis lies in modeling a meta concept to describe the notion of software history as it is present in representative tools for release history. The model will be conceptualized in UML 2.0 and implemented in Java with the use of Hibernate[Hib05].

The s.e.a.l. research group conducts a software evolution project, where the release history meta model, developed in this thesis is a base part of. The release history meta model is developed conceptually in this thesis. The actual implementation of the meta model is focused on the implementation of the issue tracking aspect, since the meta model incorporates the issue tracking domain as well. The release history aspect was implemented in the scope of another project[6] in the s.e.a.l. research group. Thereby, a release history model was implemented, on the base of CVS. The tools used to implement the CVS data model are used to implement the issue tracking model in this effort, hence, the implementations, both of the CVS data model and issue tracking model are very closely related to a possible implementation of the release history aspect of the meta model.

Keywords: Meta model, conceptual world, release history, issue tracking.

[1] www.subversion.tigris.org
[2] www.nongnu.org/cvs
[3] www.bugzilla.org
[4] www.ibm.com/software/awdtols/clearcase
[5] http://msdn.microsoft.com/vstudio/previous/ssafe
[6] The versioning data model was implemented as part of the evolizer project in scope of a internship at the Institute for Informatics

Contents

List of Figures

Chapter 1

Introduction

The importance of software evolution became more and more pertinent to software systems during the last twenty years, even if for very practical purposes such as undoing changes. The awareness for software evolution was present some decades ago. In the 80's, early works on software evolution [LB85] stressed the importance of modeling and conception of versioning systems. First versioning systems, such as SCCS or RCS, had rather simple, text-based algorithms to store successive versions of files. In recent years, considerable research in the software evolution domain brought up novel concepts and implementations in release history systems. The notion of history shifted from managing single files to change or configuration management, that is, to history management for software products as sets of software components (files).
Recent development brought up release history systems, which, as we will discover later on in the thesis, appear to have a similar notion of an objects history -we deliberately use the term object instead of file, since todays versioning systems are capable of managing different entities (files, directories, code fragments, etc.). Namely, a revision or version of an object and information, such as who, why or when a new revision was made, represent the core concepts in release history systems, no matter if they are open source versiosning tools or CMS (Configuration/Change Management Systems). However, all those systems handle these concepts differently. The information storage and data representation differs from system to system. Some of them are focused on managing single files, other on managing directories and files. Others even manage objects with no distinction of versions but with hashing algorithms. In terms of conceptual development, each of these systems has its own conceptual notion of software history -each system has a particular descriptive semantic framework. It figures, that a possible interoperability among those systems in terms of data interchange becomes more difficult as the wealth of information and degree of specialization increase[1].It would thus be interesting to compare different release history concepts in different systems. Furthermore it would be interesting to bring up a universal description for at least a group of release history systems, not the least for the sake of data evaluation and better insights in release history. So far, there has been little effort in conceptualizing such a framework. The difficulties lie in multiple aspects. If we take a data model of a versioning system, designating it as set of concepts, such as version, modification, branch, etc., then each data model has its own description rules -its own semantics. Further, each data model has a different notion of the elementary unit it applies versioning to. Some data models, as mentioned, consider files, other directories, again other consider code fragments, such as methods or classes as their elementary units.

[1]The problem of interoperability is mainly present in large, heterogeneous software environments where it is probable, that multiple versioning systems are being used in different divisions.

1.1 Contribution

In our designing effort, we bring different kinds of history management systems together and incorporate their notion of history and changes under one framework. This framework will consider the data models of different kinds of versioning systems and will provide semantics to describe each data models structure and dynamics. It is, thus immanent to mention, that the effort of developing the framework can be referred to as *development of a meta model for release history systems*. Designing the meta model for release history systems will enable a consistent information integration of different versioning systems into one data model. Thus, a reliable base is being made to consistent and valid modeling of different kinds of release history concepts and integration of versioning tools. In order to construct a meta model of the described kind, some data model reference has to be designated. For this purpose, we have considered the most common release history systems, both pure versioning systems and CMS. For the base-versioning systems, the data models of CVS[Ced05] and SVN (Subversion)[BCS05] were taken since they represent the most sophisticated and well known versioning concepts. In the class of CMS we have taken the data model of Rational's ClearQuest. The consideration of the mentioned systems brings the fact close, that we intend to develop and implement a meta model based on systemic data representation of multiple, designated tools, and not to elaborate possible meta models based on a high-level notion of release history. The effort of constructing the meta model in this thesis is thus both conceptual and implementation oriented.
While constructing the meta model, we clearly focus on the release history aspect. We further extend this aspect by adding problem reporting concepts. The intent is to broaden the information set and adding the ability to manage issue tracking data models as well. Since issues or bugs are closely related to versioning -the relation will be explained in one of the further chapters- it would be a logical step to combine these two aspects. As a base for the issue tracking data model, the Bugzilla[2] issue tracking system was taken.

1.2 Outline

The introduced modeling effort starts by introducing related work, conducted in the domain of meta modeling software evolution in chapter two. These works present similar efforts, yet different approaches and notions to history of software. Chapter two further addresses methods of (meta) modeling and draws a line of reference to the meta modeling approach applied in this thesis. Further, some important and applied UML modalities are being introduced to designate the modeling technique. Chapter three introduces different versioning and issue tracking systems and their data models. It represents the introduction of the point of reference for the construction of the meta model. By introducing the data models we lay down a set of base constructs for the conception of the meta model. After designating the reference for the construction of the meta model, chapter four starts with the conceptual part of the work, the construction of the meta model using UML2.0[Fow04]. Chapter four first starts with the introduction to meta modeling. After the introduction, the actual meta model is being derived by referencing to the CVS and SVN data models and plotting and elaborating each entity of the meta model and its relations. The construction of the meta model is done in three steps. First, the meta model is derived from the versioning data models at a similar abstraction level as the data models are on -the file/directory entity is the elementary versioned object, hence represents the lowest abstraction level. The second step extends the release history meta model with issue tracking data. Hereby the linkage

[2] www.bugzilla.org

between the two models in in focus. Having the combined meta model in place, the next step focuses only on the release history aspect of the meta model. Hereby the abstraction level is being altered in a way that enables the meta model's semantics to describe smaller objects than a file or directory. The file entity is being further specialized into smaller fragments, such as classes, methods or attributes. The purpose of this change in abstraction level is to enable the modeling of so called fine grained versioning systems, that focus particularly on source code objects.
Chapter five is about the validation of the release history meta model with another tool, that is not an element of the base set of tools used to derive the meta model. The validation helps underlining the effectiveness of the meta model in its role as a descriptive framework for release history systems. As the validation reference, the data model of the ClearCase CMS [Rat03a, Rat01] is taken. The validation is conducted by first introducing the ClearCase data model. Than a comparison of abstraction level, entities and their relations is made and discussed. In the end, chapter five discusses some interesting ClearCase features that might extend the conceptual world of the meta model enabling a better modeling of such systems as CleasCase or SourceSafe. CMS pose a broader view on release history, since they incorporate process and work-flow management and other organizational tasks besides pure version keeping.
In chapter six, the implementation of the issue tracking aspect of the combined meta model is being introduced. To conclude the work, chapter seven elaborates the experience that resulted after and during the work as well as it does outline the most important steps and problems that emerged during the work.

Chapter 2

Background

2.1 Related Work

Before starting the effort of constructing the release history model, and as a means of outlining our approach relative to other endeavors in the domain of release history modeling, we introduce efforts done in a similar way as the work of meta modeling done in the scope of this thesis.
The related work section has its goal in introducing efforts in modeling release history as a concept using meta models as a description facility. It is clear, that each of these efforts has a different goal and modeling technique and that they might not correspond to modeling efforts conducted in this thesis. However, this chapter describes each modeling effort in relation to our meta modeling approach for the sake of comparison and idea exchange.

A first work in modeling release history introduces HISMO[TG05], a meta model centered around the concept of release history. The work states the importance of an effective meta model to enable modeling and analyzing software evolution. In [TG05], history is defined as a sequence of versions, that are immanent to the kind of objects present in source code development. Objects such as packages, files, methods and all other possible entities are considered by the HISMO model. The notion of history is spread across entities of different hierarchical levels, which leads to a modeling of structural entities. Basically, the number of lines of code changed indicates the evolution of a code fragment in a class history. The HISMO model is based on the FAMIX meta model [SD01]. HISMO is implemented in a tool called Van which is a part of the Moose[1] re-engineering environment. The HISMO meta model has an important base thought, that corresponds to the motivation of our meta modeling endeavor: The importance of a meta model to effectively model the history of files in different release history systems. While the HISMO meta model is more concerned with the notion of history on a line-of-code-level, we are more interested in the dynamics of a release history data models structure. The notion of history is immanent to our meta model and taken as existent and homogeneous in its definition across multiple release history systems.
Another interesting work[PH], that takes release history into consideration is in a strong relation to OMG's[2] MOF (Meta Object Facilities), stressing the creation of a distributed versioning system suitable for MOF. The work further states, that common versioning systems are not sufficient for MOF.
The notion of release history is understood as management of multiple versions of software entities. The meta modeling aspect of this work is closely related to MOF. MOF defines an abstract

[1] A collaborative re-engineering platform designed by the same authors that designed HISMO
[2] www.omg.org

framework for defining and managing technology-neutral meta models. The goal of the distributed model in [PH] is to propose a versioning model that takes into account the distributed character of MOF. The distributed versioning model solution is based on location identificators and sequence numbers combined with rules for successor and branch creation.
The work on the distributed versioning model for MOF presents another aspect of release history meta modeling. While the distributed approach focuses on a meta model that is capable of handling a distributed system structure according to MOF and based on the framework of the MDA (Model Driven Approach), our modeling effort implements a release history meta model based on several common and well established versioning systems. The MDA framework is considered, but it is not stressed in the design of our meta model.
The HISMO meta model is taken as a related modeling effort to ours, since, first, it is concerned with the concept of release history. Further, it uses the meta modeling concept to accomplish the task. The meta modeling is done based on a certain established meta model (FAMIX). Unlike HISMO, our release history meta model exists, as mentioned, on the base of several versioning systems, that have concepts and semantics of their own. However, the two models go in the same direction of meta modeling release history, but they are based on different initial models and systems.
The distributed versioning model for MOF is taken as related, since it is concerned with the MOF and MDA, which in turn, present the newest approach in modeling and meta model concepts and which we take into consideration as well. The possible differentiation would be, that the distributed versioning model uses MOF as a reference, not taking the particular dynamics of versioning systems into focus.

The next work is concerned about the connection between release history and issue tracking . The work in [FPG03] addresses the problem of insufficient support for data analysis of software aspects. For the solving of the mentioned problem, the approach is based on populating a database that combines data from versioning as well as bug tracking and adds missing data such as merge points for versioning systems. The idea is to retrieve relevant and meaningful views of the evolution of a software project. The retrieval of data is shown as the execution of several representative queries for software evolution analysis. The approach is applied on a large Open Source project such as Mozilla[3].
The work, as in [FPG03] is the most closely related effort to our meta modeling approach. The release history meta model in our thesis too applies the combination of release history and issue tracking in one data model and the population of one database with the combined data. While the implementation in [FPG03] uses a SQL Database and scripts for data retrieval, the approach in this thesis is implemented using Hibernate for database-table creation and information retrieval. Further, the release history aspect in [FPG03] is based on CVS as an information source, whereby a meta versioning model is used in our approach.

The work in [JB05] introduces Kenyon, a system designed to facilitate software evolution research. It provides a set of solutions to problems such as the source-intensive extraction and efficient storage of analysis-specific facts, such as commit meta data. Kenyon supports release history systems that perform the analysis of a series of related layers that comprise a time-based software development history. The aim of Kenyon is to reduce the start-up time associated with software evolution research by providing a framework where new analysis methods can use any supported source code management systems and any supported data type.
The relation to our meta model is present, since Kenyon presents a meta model approach to release history related to common versioning systems such as CVS or SVN and it endorses the notion of history in software systems as a set of related "facts" (in our case revisions).

[3] www.mozilla.org

Other related work focuses on different aspects of versioning and uses different approaches and techniques[Cap03, Cap04, Jaz02, ML02, XW04, TZ04]. Some are concerned about visualization of software evolution[TB96, CC03], whereby others take the focus in prediction of change propagation in software systems[HH04]. A common base for all these approaches is the (release) history as a concept. The aspects, under which this concept is modeled and understood, differ however.

2.2 Modeling concerns

2.2.1 Meta modeling

The word "meta" comes from the Greek and means "further on" or "beyond" in a free translation. The definition of meta modeling does not exist in a strict, universal form. It is a concept applied in many scientific and real life endeavors, and each time it is used in the frame of a certain concept. The goal in this short digression, is not to provide a profound definition of meta modeling or a meta model, but to rather describe this concept in the context of out work of designing the release history meta model.

When designing a complex software system, developers are often faced with different interdependent concepts in form of programming languages, platforms, etc. Further, each of these concepts has its own conceptual world[Fis05]. Each conceptual world exists on different abstraction levels, which have to be compatible in a sense, to enable the developer to combine these different conceptual worlds in order to develop the system. The problematics here are, that each conceptual world has its own semantics[4], and a developer has to deal with all of them, for each conceptual world, in order to develop. It is thus a necessity, to have a sort of universally applicable framework to be able to describe and define the particular concepts, their relations and their notation. So, it is guaranteed, that the different concept worlds ca be put together consistently and effectively. The necessity for such a framework comes also from the MDA (Model Driven Approach) where the different abstraction levels of concepts play an important role. There are different kinds of such frameworks or notations of concepts and their relations. One of them is the application of a meta model.
A meta model describes a conceptual world; that is, the structure of the particular concepts and their allowed relations in form of a directed graph, whereby, the knots of the graph represent the concepts and the connecting lines represent the feasible relations. The application of a meta model follows the object-class paradigm[5]. Hereby, a meta model describes the concepts based on their similarity and not from case to case. The description of concepts generates meta data , based on which a concept can be defined. This meta data can be also considered as a classifier for a concept and its instances. A classifier can in turn be an instance of a higher-level description (classifier). This higher-level classifier would then be a meta-meta model. Meta modeling addresses the problematics of information transition from one abstraction level to another. Hereby, the problematics in particular lies in different abstraction mechanisms[6] concerning information.

To conclude this short introduction about meta models, the following can be stated: Meta modeling is the application of valid frameworks to describe the semantics of different conceptual

[4]Semantics considered as the full definition of a conceptual world, whereby structural aspects are defined by static semantic, and behavioral aspects are defined by dynamical semantics.
[5]A class describes a set of objects. Examples are, for instance, grammar-word, template-document, package-class, etc
[6]Examples are generalizing/specializing, information hiding, organization, structuring/destructuring, etc.

worlds on different abstraction levels, whereby it is possible to layer the frameworks used for description. A meta model is a way of applying meta modeling in form of a directed graph. Meta modeling also addresses the problematics of information handling across different abstraction levels.

2.2.2 UML: applied naming and style conventions

The modeling of our release history meta model is done with UML 2.0 using Microsoft Visio 2003 as the graphical editor. In order to complete the description of the modeling approach in this thesis, it is a necessity to point out the conventions in notation and styles of UML for this task. The goal is however not, to give a profound description of the UML super- or infrastructure, but rather to describe the applied naming and style conventions. The reason why this description is done, is, because, there are no binding, overall applicable notation style guidelines in UML we can refer to and thus we point out the conventions used in this thesis to underline this particular notation approach. Modeling efforts can deviate from one another in notation or styles. All the different styles and notations must however be UML -standard conform and be applied according to the semantics of the UML.

According to the OMG[7]'s specification for UML[8] and other resources [Fow04] the following conventions in notation were used in this thesis:

- UML -class diagram: Figure 2.1 shows the common full notation of a UML class diagram. Class name and attributes are used; operations on the other hand are not used, since the structure is in focus and not the dynamics of the models.

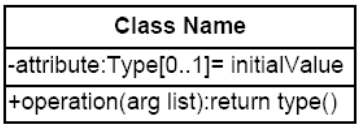


Figure 2.1: UML 2.0 class diagram

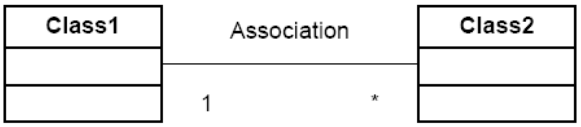


Figure 2.2: UML 2.0 bidirectional association relation

- UML -class relations and their multiplicity: In all the data models in this thesis, most of the relations are bidirectional. In some notations, an association relation is drawn with arrows on one or both ends to denote, whether the relation is uni- or bi-directional. In the notation style used in this thesis, a bidirectional association relation is drawn without arrow-ends (Figure 2.2]. An unidirectional association is drawn with an arrow-end, pointing out the direction of the relation. All other types of relations (aggregation, generalization, etc.) are

[7]Object Management Group; www.omg.org
[8]Unified Modeling Language

drawn according to the common UML notation standard.

The multiplicities of the relations are denoted according to UML standards in the following way: Class 1 in Figure 2.2 is associated with **many (zero or more)** instances of Class 2, whereby Class 2 is associated with **exactly one** instance of Class 1.

Chapter 3

Release History Systems and Bug Reporting Tools

This chapter covers the various release history systems and their respective data models for version control. First we will give an introduction and an overview of the concepts and main functionality of some of the most used versioning systems, among them the CVS and Subversion versioning systems whereas the data models of those two release history systems will also serve as a base for the release history meta model we are developing. After CVS and SVN (Subversion) we will also take an overall look at Rationals ClearCase change management and release history system. For our topic, ClearCase is especially interesting in two ways. First, ClearCase incorporates concepts regarding release history and change management, such as project team management and thus tailoring the visible and accessible versions of files to a particular group of developers , or the concept of grouping collections of files to meta entities for better managing and deployment, and so on. These concepts are not part of the, for example, CVS and SVN versioning concepts but they might be a useful addition to the meta model. This will be covered in one of the following chapters, after we have established and validated the meta model. The second way, in which ClearCase is interesting for our research is that this system is the one the meta model is going to be validated against.
Further we will introduce some other versioning systems, such as BitKeeper, Visual SourceSafe, Aegis, Arch, OpenCM, etc. used in practice. We will not examine them in detail, their data models respectively, as we do with CVS, SVN and ClearCase.
After the version control systems we will have a look into diverse bug reporting tools, especially Bugzilla. The reason, why we examine Bugzilla in detail is that, it's a common and most used opensource bug reporting tool available and, the meta model is going to be extended by adding the Bugzilla data model.

Other bug tracking and reporting tools would be, e.g. for java code, FindBugs, JLint or Bandera, for the GNU project there is GNATS (aka PRMS) and other problem tracking tools like JitterBug, Tracker or the Debian Bug Tracking System.

3.1 CVS

Among the popular and efficient release history systems there is CVS, the Concurrent Versions System. CVS is a versioning system that records the history of source files. It was first not much more than a set of shell scripts written by Dick Grune who posted them to the comp.sources.unix

newsgroup in the volume 6 release of July, 1986. An interesting fact is, that no actual code of these first shell scripts is present in the current version of CVS but much of the conflict resolution algorithms still come from the original scripts. CVS is a further development of RCS (Revision Control System); RCS is a version control system, mainly for text files such as source code files or configuration files. RCS manages only single files and thus cannot be used in projects of a larger scale. Though, CVS uses the same file format as RCS.
CVS is basically a command line program but in time there was an appropriate graphical user interface developed for nearly all current operating systems. Examples are TortoiseCVS and WinCVS for Windows, MacCVS for Apple Macintosh and Cervisia for KDE and the Linux platform.
By defining CVS as said we could stop at this point of description, because the core functionality of CVS is to record and store the history of a developers source files or any other kind of files. However, the way, how CVS stores the history of files is interesting; not just because of CVS itself but also for the purpose of designing the meta model in the later sections of this thesis. So, we will have a look into how CVS manages the file history.

CVS Data Management

Generally, CVS stores all versions of a file -we will use the term "file" hereby having a source code file in mind, because the most common file type used for version control are source code files- in a Repository. All versions of a file are stored as a single file where only the differences between the versions are stored. If a developer wants to make changes to certain files in the repository, she checks out these files to a working copy on her local machine so the base-files are left unchanged until the commit operation changes them to the most recent version.
Another concept immanent to versioning systems in general is the concept of branching and tagging files. A branch is a separate development line of a file. CVS has it own branching concept. Each time a developer wants to branch off and develop in a separate line, she first has to tag the file as a branch. A tag is a sort of file meta information, that can be attached to a file. At this point, with no actual changes made to the branch file, CVS only stores the branch point in the revision number of the branch (for instance, if a branch was made at revision 2.3, the branch point would be 2.3.2.). If then changes are committed to the branch, the first branch revision would be 2.3.2.1; then finally the branch file is stored in the repository and is treated as a separate, new file.
Releases, as sets of different revisions of files, are not explicitly present in CVS although CVS is capable of storing releases. For this and other purposes, a developer tags a set of revisions. In the case of tagging a file (revision of a file) for a release, the tag specifies which release a revision belongs to. The tag names, thus the release names can be arbitrary, free settable by the developer. So a release is actually a set of revisions of files that are tagged with the same tag, which holds the information, that these revisions of files belong to a certain release.
CVS helps managing files under version control, especially in a project, keeps track of older versions and restores them if necessary. A comparison of versions is also possible. By saying that CVS can manage Files in a project does not mean that CVS can be considered as substitute for project management and control; CVS also has no built-in process model to ensure that a developed software goes trough a set of different steps before landing in production. These specific functions are for instance immanent to Rationals software products (ClearCase, ClearQuest) which we will have a look at in more detail later on in this thesis.
Other CVS features concern for instance the repository, hereby highlighting the commit and check-in operations. Unfortunately CVS's commits and check-ins are not atomic, meaning, when a commit is interrupted, the repository is left in an unstable, inconsistent state. Concerning the repository further, CVS provides the possibility to set permissions on access to different parts of a repository (local or remote). CVS is however capable of line-wise file history tracking, i.e. for

each line showing at which revision it was most recently changed, and by whom. Another convenience in CVS is that a developer can check out only one directory out of the repository for individual development.
When developers face a conflict in a single file, most of them manage to resolve the conflict without much problems. However a more general definition of a conflict involves problems too difficult to solve without direct communication between developers. CVS cannot determine whether simultaneous changes in a single file or across a collection of files will logically conflict with each other. CVS understands the concept of conflicts in a pure textual way, arising when two changes to the same base file are close enough to "corrupt" the merge command. We have thus pointed out the base characteristic CVS features but have certainly not mentioned all of them. This is not the scope of this thesis. Ultimately CVS is, from the fact out that it is a de-facto standard in versioning, a very easily deployable system and it is very reliable (not taking the various user interfaces for CVS and their bugs into mention). It possesses the most common release history features such as line-wise file history tracking, modular repository structure and a set of other convenient features. There are some lacks as the non atomic commits or the inability to discover conflicts in a broader meaning than the line-wise interferences when merging versions.

3.2 Subversion (SVN)

The next release history tool in our overview is Subversion[BCS05]. Subversion (SVN) is an open-source version control system. As CVS it manages files over time. In addition to the CVS functionality, SVN manages directories as well. More precisely, what SVN does is, it stores a tree of files in a central repository, that can be regarded as an ordinary file server except that it records every change made to files or directories over time. Interesting here is that given SVN's architecture, the data can be optionally stored in a Berkeley database or in a common FSFS database.
The development of SVN began in the early 2000 when CollabNet[1] started searching developers to conceive a replacement for CVS. CollabNet offered collaboration software of which one part was history tracking or version control. This version control part of the collaboration software was originally dependent on CVS as its initial version control system and given some of the limitations CVS has, concerning versioning and file storage, CollabNet decided to make its own version control system from scratch. On August 31, 2001 Subversion was fully functional and replaced CVS in managing its own source code files.

Subversion Data Management

SVN is not much more different than its predecessor, CVS; it stores the history of files, a developer can check out a working copy of the files to be able to work locally, comparison of versions is also possible. However, the next paragraphs point out that SVN has some different features compared to CVS, especially concerning file and directory versioning.
One of the new things in SVN is the directory versioning. CVS remembers the history of single files, whereas SVN manages the history of files "virtually", meaning it tracks changes to whole directory trees so it manages files and directories. If a developer wants to check out a working copy, she checks out a whole or a part of a directory tree under version control by SVN.
Another issue, is the version history itself, in light of the respective versioning technique in CVS and SVN. We have stated earlier, that CVS is capable of managing (only) files. Thus, some operations like copying or renaming, that apply to files, but that could be considered as actually making changes to directories, are not supported by CVS. Also, when replacing a file in CVS with a file

[1] www.collab.net

of the same name as the replaced one, the history of the old file is inherited by the new one, even though the two files might be completely unrelated to each other. Within SVN, the mentioned operations are supported and, by for instance replacing or renaming a file, the new file comes up with a clean history.
Another relevant issue are the commit operations. We have seen that the commits and check-ins in CVS are not atomic. With SVN, a commit is considered as a set of changes (or transactions) where by the changes are first stored in a transaction tree and latest after a commit command they are being stored as a definite revision tree. Only when an executed commit operation is complete, a new revision is made out of the current transaction tree. This means, that commit operations in SVN are atomic and that it is highly unlikely that the repository could be left in an inconsistent state. Each revision in SVN is a new and updated copy of the base directory tree under version control by SVN.
A very convenient concept in SVN are the branching,tagging and release concepts. In fact, here are none! SVN is fully capable of managing tags as file meta information, branches as separate development lines and releases as sets of revisions without explicitly having a concept or a mechanism for it. Tags are common file meta data that are managed and kept for files or directories. Branches are in fact separate directory trees made out of a current main-trunk directory tree. When a branch is made, for a file, the revision enumeration continues on. The only property that changes is the path to the file or directory that moved from the main trunk to a branch. SVN tracks the changes made to both the main trunk and the branch as a log of the same file, telling the developer where a particular change (main trunk or branch) was made and whether a revision corresponds to the main trunk or the branch. The difference between a release and a branch is minimal. Again, a release is nothing more than a copy of the whole or a part of the current directory tree under version control. The only difference between a branch and a release in SVN is that a release is not supposed to be tempered with once it is designated, meaning no files ought to be changed, otherwise a release becomes a branch.
Speaking of versioning differences there is of course one obvious difference between CVS and SVN. The version numbering concepts are different. In CVS version numbers are an even number of period-separated decimal numbers. By default revision 1.1 is the first revision of a file. Each new file gets the second number set to 1 and the first number set to the highest first number of any file in a repository. In SVN the revision is a decimal number starting from 1 as the first revision of a file and increases by one for each new revision.

3.3 Rational ClearCase

We have considered two well established and widely deployed open source release history systems; CVS and Subversion(SVN). These tools designate the most common and efficient versioning concepts. We have stressed that these tools are well suited for small to middle sized projects but not for large scale projects, at least not without a well defined versioning strategy. The tools considered in previous sections go up till the level of managing changes and project planing in large-scale, corporate environments. The next step thus would be a tool that not only remembers the change history of files in a repository but also supports or incorporates the whole software life cycle process, especially the entire problematics about change management and the appropriate project management that comes with it in larger corporate project environments. Such a tool, or better, a set of tools is Rationals change management software: ClearCase (LT, MultiSite, etc...) and ClearQuest and its versions.
The ClearCase family of products also provides software asset management with version control, baseline management and build and release management. The ClearQuest products on the other hand provide defect and change tracking and work-flow support. The concept of most interest

for the meta modeling later in this thesis, is the release history aspect of ClearCase, but certain concepts we will examine, could be used to enhance the narrowish view of change or configuration management, provided by CVS, SVN and other versioning tools.

ClearCase Data Management

When looking at ClearCase's functionality in more detail we see that this functionality is a set of different concepts and processes such as: version control, automated workspace management, parallel development support, support for disconnected usage, local, remote and web client access, transparent, real-time file and directory access, build and release management, automatic backup and restore, etc.[Rat03a, Rat03b, Rat01]. ClearCase in a way exceeds the version control functionality by integrating version control but on top of it, providing a lot of other related processes, which can be thought of as supporting or widening processes relative to version control.
Nevertheless, ClearCase manages the files and their history mostly by the same concept as for instance CVS or SVN. Versioning in ClearCase incorporates creating new versions of different kinds of source files, comparison of versions of source files, branching off separate development lines, merging changes between versions, change tracking (who, when or why has a particular change been made). When versioning files, ClearCase does not overwrite a current file but stores all the versions as separate files. All files are stored in the repository, the so called VOB (Versioned Objects Database). An interesting fact is, that also unversioned objects can be stored and viewed by a developer[2].
Source code files have been stored in a repository of a respective versioning system. Releases could be separated and branches could be made. A software project notion, meaning having a project leader, one or more developer teams, a process model and so forth has not been a part of the versioning systems like CVS or SVN so far. ClearCase however, incorporates those features. So when speaking of versioning files in ClearCase we speak of projects the files are in. Under a project in ClearCase we consider a specific product of a development effort, for instance a corporate web site.
The Unified Change Management (UCM) technology is a core concept in ClearCase. In UCM, a project is represented as an object that contains configuration information (components, activities, policies) needed to manage and track the work on a product. A common UCM project consists of a shared work area and a number of private work areas for each developer. A work area consists of a view and a stream. A view is a directory tree, that shows a single version of each file in a project (we have seen, that SVN manages directory trees too). A stream, as shown in Figure 3.1 is a ClearCase object that contains a list of activities and baselines and determines which versions of a file appear in a view. An activity (Figure 3.1) is also a ClearCase object that consists of a change set (a set of files) that a developer creates or modifies. A base line, as shown in Figure 3.2 holds one version of each file in a component. It represents a version of a component at a given stage in project development.

3.4 Other versioning tools

This section encompasses further versioning tools to underline the modeling approach that takes the CVS, SVN and ClearCase data models as a base for constructing the meta model, by showing that the systems to be considered also do have a similar information base as the CVS, SVN and

[2] In ClearCase terms, a team member

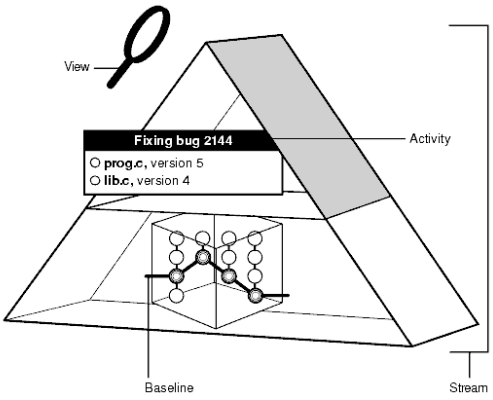


Figure 3.1: ClearCase stream overview

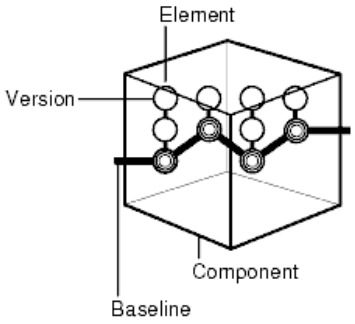


Figure 3.2: ClearCase baseline object

ClearCase models. Thus the meta model, which relies on the data and information of the base-models, is theoretically applicable to all upcoming systems in this section, i.e. their data models. Ergo, the meta model's relevance becomes more pertinent and the model itself stays valid for a larger scope of release history systems.
The goal is not, to go into the particular systems and describe their data models, but to give and overview of the particular concepts and structures that lie beneath each system, whereby we will take the liberty to point out the similarities between the considered systems and those to be described in this section in order to endorse the idea of applicability of the meta model mentioned above.

By looking at specific release history systems, we have already seen, that those systems can ruffly be divided into two classes. One class is represented by such systems as SVN, CVS, Arch, Monotone, etc. These systems represent the simple versioning systems with no integrated workflow or process management (not, or hardly applicable for large scale, corporate change management). The second class has systems like Visual Source Safe, Bitkeeper or ClearCase. Those systems can be considered as change or configuration management systems with extensive versioning capabilities. We will continue in the same matter, and describe such systems as BitKeeper and Source Safe, which belong to the second class of release history systems, and tools such as Arch and Monotone, that belong to the first class of release history systems.

3.4.1 Visual Source Safe

Visual Studio from Microsoft has introduced Visual Source Safe for managing the history of files across multiple projects and developer teams. This tool belongs to class two [3] of our release history systems . Visual Source Safe (VSS) is set out as an additional tool to the Visual Studio .NET for managing the version history of both text and binary files.
VSS has a typical structure regarding data storage. Native files (master copies) are stored in projects in a VSS database. A project is not more than a set of files. A project can be shared among different developer teams and cross-platform. VSS copies a file, which a developer wants to edit, from the database into a working folder for that developer. Interesting is, that VSS makes a distinction between the two file types mentioned, namely, text files are those that contain only characters grouped in distinct lines. Binary files represent all other file types. The idea behind is, the separate treatment of older versions (states) of a file in terms of version history management and reconstruction. VSS can reconstruct an earlier state of a binary file, but can not display it. For most operations, text and binary files can be treated the same.

The check-out- check-in concept in VSS is pretty much the same as in, for instance SVN or CVS: when a developer wants to check out a file, VSS copies the file into the working folder of the developer. She can now apply changes to the file. Usually, check outs follow a single-check-out-policy, meaning that if a file is already in use, no one else can check out or commit changes to that same file. A single checkout policy is permanent for binary files. If a user only wants to read a file, she does not have to entirely check out a file from the database; instead VSS offers a GET or VIEW FILE function for that purpose.
The versioning concept in VSS is extended by some additional information, that is used for version control and history services. To track a file, VSS uses three methods, or three types of information for that matter: *version numbers*, whole numbers that increase for each new version of a project or a file (here we see the similarity to SVN, where a whole number is used as aversion number, a project in VSS as well as a file can have version numbers, which corresponds to the directory- file relation in SVN), they are internally managed and assigned to files by VSS, completely transparent to the user; *labels*, which are simple strings up to 31 characters that can be attached to every version (here again a similarity to CVS (SVN): labels can be considered as meta data or properties to a file or its revision); *Date/Time stamps* that tell the time, a file was last modified.
When branching a file in VSS, the file is being taken into two separate directories (paths or branches, according to the VSS documentation) at once. As in SVN for instance, the path to the (branch)-file is changed, relative to the path of the file the branch is made from. VSS tracks the history of branches under different and distinct project names. The two files (the file in the current project and its counterpart in other projects) have a shared history up to the branching point, and divergent histories afterward.
When merging files, VSS provides two methods: visual merge and manual merge. VSS can not resolve conflicts, instead it offers the developer the possibility to manually resolve those conflicts. In a short digression, we state that resolving conflicts on binary files, in the terms of VSS, is not quite an easy task, since a binary file has no clearly defined, distinct lines of characters with explicit line delimiters. Merges occur in VSS in three circumstances: when using multiple check outs, that is, when multiple users check out a file, the subsequent user's changes are combined with all other changes (the first user's changes, since after multiple subsequent check outs she simply checks in the file), when explicitly merging previously branched files - hereby the changes made in one branch project are merged with the changes in an other project,and when getting a file. In any merge, what happens is the same: VSS takes the differences in changed files, compares

[3] Change or configuration management systems

them to the original file then creates a resultant file with all the changes.
Additional interesting features in VSS are, for instance shadow folders. Shadow folders are centralized folders on a network server that contain all files in a project; a sort of a centralized area to view and compile source code. More precisely, they contain the most recently checked in version of a file in the project. Shadow folders are optional and serve in generally two situations: to allow a user to view, but not modify the files, especially, when that user does not have access to VSS; and to prevent having a compilable copy of a project in a local working copy.

3.4.2 BitKeeper

BitKeeper[4] is another versioning tool that falls into class two of our categorization of release history systems. It is a tool for revision control of pure source code. It builds up on many concepts known from TeamWare - later called Forte TeamWare then Forte Code Management System, which is a revision control system for source code, developed by Sun Microsystems. TeamWare introduces some new features in contrast to CVS or RCS, such as hierarchically structured repositories or atomic updates of multiple files (as present in SVN or Perforce).
BitKeeper, like some other change management systems, enables developers to work concurrently on the same project. It was also designed to support globally distributed development - when looking at the architecture in high level terms, taking the TeamWare underlying concepts also into consideration, BitKeeper works as a system of files accessed by client programs, disconnected operation, change sets, etc.
An immanent concept of release history systems is the repository or database, the files are stored in. A BitKeeper (BK) repository represents a collection of files, sometimes called a *"tree"* or just *"repo"*. In contrast to other versioning systems, such as CVS or CleasCase, BK's repositories are self- consistent units, that incorporate all necessary functionality to perform development and versioning work. This is an interesting concept, since, usually, versioning systems have one central repository, where a user can make working copies of just a part of the repository. In BK, a developer makes a copy of the entire repository, called a *"clone"*. So, a developer can alter, even delete, her own repository thereby not affecting a shared repository or repositories of other developers. The relation between a repository and its clone is a parent -child relation, meaning, that BK remembers the parent repository as such. Thus it is straightforward, that there must be a sort of hierarchical repository structure. Changes that are made, propagate between parent and child, but also among multiple child repositories. Another concept, very similar to ClearCase, is the concept of Change Sets. A change set is a grouping of related changes to files, and the interchange medium among BK repositories.

BK manages the following three file types: text files (e.g. source code); binary files (images, text(word) documents); symbolic links (Unix). For these file types the following information is versioned: file contents, filenames, file flags, file permissions. The revision number is a set of two comma separated integers, whereby the second integer increases for each new revision. When branching, a developer effectively clones the repository. As in CVS or SVN, a file can have some additional meta information attached to it. This meta information comes in form of tags. Tags are symbolic markers that identify the state of a repository in a certain point of time. They are also used to more easily refer to a certain release.
When trying to view or restore to an earlier state of the repository, BK can use multiple sources of information to do that. A developer can specify an older revision of a file, change set or tag level. If an older revision of a file is needed, the file's revision is needed to specify the revision that is needed. The same goes for the change set rollback: the appropriate change set revision is needed.

[4] www.bitkeeper.com

As for tag level rollback, only the name of the particular tag is needed.

3.4.3 GNU arch

The next revision control system belongs to the first class of versioning systems discussed in this thesis. GNU arch[5] (Ga) is an open source versioning system with some interesting features, not present in most other versioning tools. However, it follows the same concepts of versioning as CVS, SCCS or SVN do.
Concerning the versioning of objects (file trees), Ga uses a somewhat different concept. Namely, each revision in Ga is uniquely globally identifiable. This sort of versioning allows merging and application of changes from completely disparate sources - unlike most other versioning systems, where merging occurs mostly in the same repository (database) and among similar projects or inside one project. Further, Ga is a scalable, decentralized system without any central servers and repositories; this removes the need to be authorized as a developer to a server in order to work with Ga. The concept is rather, that a head developer makes a read-only copy of the entire project (via HTTP, FTP or SFTP), and each developer can acquire that copy, make changes to it, then publish her change set so that the head developer can manually merge the changes into the head project and update the read-only copy. However, if one wants to simulate a centralized system, the head developer could allow SSH or write access (FTP, WebDAV) to a server, enabling only authorized users to commit changes.
Further, Ga is capable of atomic commits. A source tree must be in consistent state before a commit can be executed and generally, commits are not visible until executed completely. Thus, if commits become interrupted, they remain invisible and have to be rolled back before additional commits are executed.
Ga supports change sets, thus, instead of tracking individual files. Each change-set can be considered as a snapshot of a source tree. Here again the similarity to other versioning tools, such as SVN, where the versioning takes place at a directory (file) -tree level, rather than on a per-file level is given. The same goes for branching- a branch is handled as a tag; it declares an ancestor revision, and further development continues from there.
A common problem in versioning tools is the renaming or moving of files. With Ga, files and directories can easily be renamed, since they are tracked by an unique ID rather than names. Thus, the history of a file is preserved and patches to files are correctly merged despite the changed names (even across different branches). Another interesting feature, not encountered as such in the previous release history tools, are cryptographic signatures. Every change set is stored with a hash to prevent possible corruption. These hashes can optionally be signed (GnuPG or PGP) to avoid unauthorized modification of files.
Gnu arch is still a maturing project, concerning eventual serious problems on portability to non Unix platforms, and it is not so easily learned as some other versioning tools. Mostly because of the arch specific commands, which could be intimidating to new users and thus need some initial learning time.

3.4.4 Monotone

Another tool similar to GNU arch is Monotone[6]. It's an open source revision control system, with a similar distributed approach to managing files in repositories as GNU arch- the interested reader is encouraged to recall, that GNU arch is capable of managing multiple "stand-alone" repositories and the interactions among those repositories- merging and branching into and out of disparate

[5] www.gnu.org/software/gnu-arch/
[6] http://venge.net/monotone/

projects (repositories).
Before continuing the description of Monotone, we first point out some interesting features of the tool that didn't appear in versioning tools so far. First off, Monotone uses SHA-1 (Secure Hash Algorithm; cryptographic hash function, the successor of MD5) hashes to identify files or groups of files instead of revision numbers. Another, monotone specific feature, is the use of netsync[7] for synchronizing trees (remember the distributed approach to managing file trees and repositories). Netsync is a custom protocol, considered to be more robust than most other network protocols.
It was mentioned, that Monotone stores a hash instead of a revision number for a file. The concept of versioning, i.e. the distinction between the different revisions of a file or a file tree, can be described in a parent - child relation between the native (parent) file and the newer versions (children) of that file. The relation between a parent and a child file consists of the edit, that was done to the parent file and of which the child file was created. In managing and storing different versions of a file, Monotone can either store a complete copy of the native file, or, since successive versions are often very similar, store only the difference between two consecutive files.
Versioning in Monotone is not only limited to files. A developer is also capable of taking a snapshot of certain files in a collection. This snapshot is referred to as a file tree. So in Monotone, one can also manage entire file trees. The advantage of that sort of versioning is, that changes can be, for example, reverted for multiple files at once. In order to make a snapshot of a tree, a manifest file (plain text) is being created. The files content consists of plain text lines divided into two columns: the first column holds the SHA1 codes (revisions) for each file, and the second column holds the path to the file.
In Monotone, branches are designated across multiple files. Every file in a branch has a reserved branch id. Branches can be given symbolic names to make it more easy to distinguish them. A similar concept to the branching in Monotone, is the use of tags (tags as in SVN, CVS, etc.), where as in Monotone, the so called branch cert is a unique identifier for a set of files, separate from the optional symbolic name. It is said before, that the the relation between revisions can be thought of as a parent - child relation, thus, a tree structure. In a branch, the revisions with no child revision on them, are called the *heads* of the branch. Monotone can automatically attempt to merge the head revisions in a branch. If a conflict arises or another reason, why a merge cannot be executed, Monotone leaves the branch in a consistent state with no changes made.
Despite some interesting concepts, that Monotone offers - e.g. SHA-1 hashes, distributed repository management, certificates- the question remains, if these concepts are really scalable for larger projects. For instance, an efficient certificate management for each file's history in such large projects remains questionable in terms of usability - for comparison, ClearQuest, as a large scale change management system manages the history of files without any certificates.

3.5 Bug reporting: tools and concepts

In the previous sections we have looked at release history and introduced the tools and concepts that are used to manage history of files or file structures. We have given an overview of different classes of versioning tools and pointed out some specific features immanent for a particular tool. Keeping track of source code history and conducting change and configuration management based on the history of development can be considered the most important concepts in the release history domain.
However, during development of a source code project, certain problems or dis-functionalities in the software may arise. Such problems can emerge from one revision to another and are referred to as "*bugs*". Thus, bug tracking is a necessity and a nifty addition to release history management.

[7] A network protocol

Further, bugs (issues) can emerge from modifications made to a file. Considering, that the modified file is under version control, the bug emerges out of that edit, that is, from the modification report of that file. Thus bugs can be considered as an additional, structured information to a modification report.
When considering bug tracking, we understand the storage and management of issues related to programmatic or even systemic instabilities, faults or conflicts during development. Usually, these issues are stored in a dedicated bug (issue) -tracking system. These systems mainly consist of a database (open source or proprietary) where the data to a specific issue is stored, and the client and administrator side access layer (usually web based access, like Bugzilla). Other bug tracking systems can also be part of a larger CMS (Configuration Management System), such as ClearQuest, which is a part of the Rational ClearCase family of software products. As well as for versioning tools, we can make the distinction between "pure" bug tracking tools (GNATS, Bugzilla, JitterBug, etc.) - web based, open-source, free accessible- and proprietary bug reporting tools as integrated parts of corporate CMS (ClearQuest, etc.). The distinction here will not be necessarily stressed as for versioning tools. The reason is, among other, the rather static structure of a bug reporting system. Static in the sense, that bugs cannot be merged or branched of, they don't have a history in the sense of revision history. The static structure enables, thus, a more general approach to the description of bug reporting tools without loosing relevant information. However, if there is obvious difference between bug tracking tools, it will not be hesitated to point that difference out.

This section will cover the introduction of several well known bug tracking tools in the manner as in the previous section. Three characteristic bug reporting tools will be discussed. In our following description, both web based, opensource bug tracking tools (Bugzilla and GNATS), as well as CMS bug tracking tools (ClearQuest) will be introduced.

3.5.1 Bugzilla

As mentioned above, Bugzilla [Tea05] falls in the class of free, web-based, open-source issue tracking tools. It is the most common web based tool to manage bugs. Initially it was used to manage issues in the Mozilla Foundation[8] projects; by now, external projects, both open source and proprietary, can submit their bug reports too.
The architecture of Bugzilla as a tool is rather simple. It requires an installed server and a database management system (PostgreSQL or MYSQL, etc.) to be operational. Further, Bugzilla requires a suitable release of Perl 5 along with a set of Perl modules for the installation and a mail transfer agent, such as Sendnote [9], qmail[10], Postfix[11] or Exim[12].
Bugzilla as a concept is pretty much straightforward. The bug (issue) is the center of the concept. All other information is concentrated around an issue. As mentioned before, a bug tracking system is rather of a static nature. Bugzilla is not much different, since bugs cannot be merged, branched-up or versioned. However, bugs can depend on or block each other, they can be in different states, depending on their severity or priority. Further, in Bugzilla, the notion of a "bug" is taken more generally; it is not strictly bound to an programmatic fault or conflict within a software module. For instance, mozilla.org uses Bugzilla to track feature requests as well.
A bug in Bugzilla follows a strict work-flow -also called the bug life cycle (Figure 3.3). When a bug is submitted, it enters the state "new " as either confirmed or unconfirmed. Then it is being

[8] mozilla.org
[9] http://apgap.com/pub/SendNote.html
[10] www.qmail.org
[11] www.postfix.org
[12] www.exim.org

assigned to a developer. When the developer has resolved the bug, it can either be verified, if the solution worked out or it can be reopened if the solution was not satisfying. If a bug is verified it is being closed.

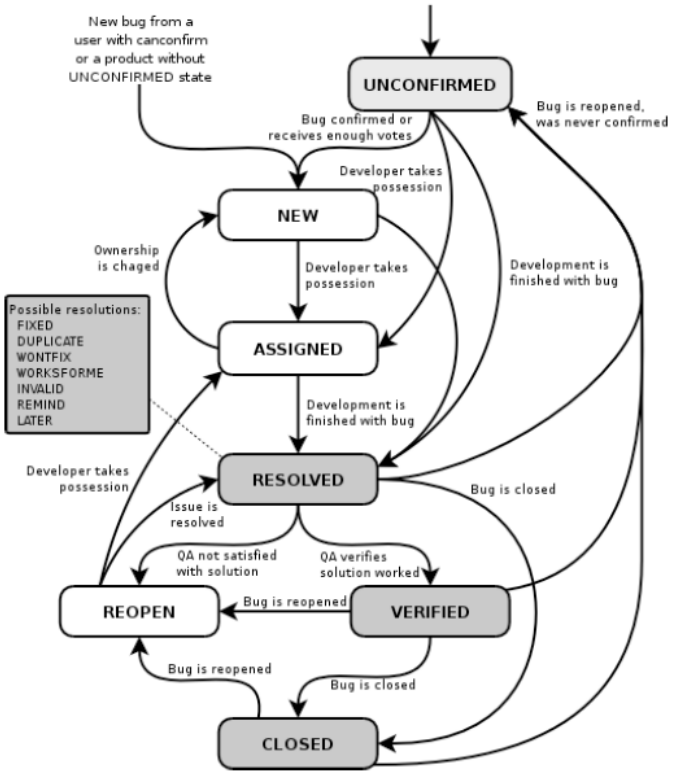


Figure 3.3: The life cycle of a bug in Bugzilla

This life cycle is currently hard-coded into Bugzilla. It manages the entire work-flow for a bug and defines clear states a bug goes through. We will examine the Bugzilla concepts in more detail when we elaborate about the underlying data model in one of the following sections.

3.5.2 GNATS

GNATS[13] is a web based GNU bug tracking tool. It is designed to be used at a central support site, where users can communicate problems over e-mail, or a web based client that is communicating with the GNATS network daemon. GNATS was designed as a tool for maintainers, unlike Bugzilla, which is a free accessible bug tracking system for developers, maintainers and users.
In GNATS the bugs (issues) are addressed trough problem reports. These problem reports are grouped in context-defined problem categories and are stored in a database, set up to archive and index those problem reports. GNATS actually has the role of an archive for field separated textual data.

GNATS further automatically notifies responsible parties of possible bugs and organizes problem reports.

[13] www.gnu.org/software/gnats/

3.5.3 Rational ClearQuest

Rationals ClearQuest presents a CMS integrated problem reporting system. Unlike Bugzilla or GNATS, ClearQuest is a problem tracking management system for issues and change requests as well. It further incorporates an entire workflow management process and is a part of a larger application suite. It however manages issues in a similar way as, for instance Bugzilla, meaning, that the data model reflects certain similarities ti the common issue tracking system. Due to lack of proper information, the ClearQuest system cannot be extensively discussed. However, we've introduced ClearQuest as a counterpart for the web-based issue tracking systems to emphasize the distinction of "common", web-based, pure issue tracking systems and the similarity in the issue tracking concepts of these two sorts of systems as it was done for the various release history tools.

3.6 CVS, SVN and Bugzilla: The data models

The next step in the versioning systems discussion is to look a bit closer at the most representative versioning and issue tracking systems and see how they actually manage the data. For this purpose we look at the respective data models the particular systems have underneath. The data models are the first fundamental step into shaping the base for the meta model. By assessing the data models of the various systems we get a comprehensive base construct to rely on while developing the meta model.
For the construction of the data models we will use UML 2.0. Modeling details, such as naming and multiplicity conventions and their variations in UML were pointed out in the chapter two, in the section "Meta modeling".
The approach to making the data models has come from the server side. Hence, while examining the data we've looked into how the data is actually stored in the repositories, whereby the release or file logs for instance, stored in the repository were a great help in retrieving the relevant data. The retrieval of data is divided into basically two main steps. The 1st step was to define the relevant data to be extracted. This data contains information such as revision and release numbers, tag names, branch information, commit messages, locks, bug IDs, bug states, etc. There is always a trade-off between data we want to put in a data model and data that is maybe nice to have as information but is not really necessary. When developing the meta model later on we will see that this trade off gets even more significant because there we will also have to deal with information that might be too detailed or to specific [14] to be put into the meta model.
Once the relevant information has been extracted, the 2nd step was to designate and separate the extracted relevant data to classes in a UML diagram. The separation into classes was based on the data representation in various tools for the particular systems. Tortoise SVN was used to retrieve the data representation from the SVN versioning repository and WinCVS and Tortoise CVS were used to extract the data for the CVS versioning system. For the issue tracking systems, Bugzilla's data model was represented by the bug-report web page on mozilla.org. Given the particular tools there were some slight differences in the data representation. For instance, the Eclipse CVS plug in showed a different log entry of a file than WinCVS; symbolic names were represented differently (e.g. in WinCVS a symbolic name was noted as "*1.2 : test release*" whereas in the Eclipse CVS plug in the description came first and the version as second).
Despite the mentioned differences in data representation a consistent data model was derived for CVS, SVN and Bugzilla. The next subsections describe the respective data models in detail.

[14] The problematics of merging data of different data models into a meta model implies a more high level view, a less detailed view, if you want, of the merged data

3.6.1 The CVS data model

This section describes the CVS data model. The description consists of two parts. First, a detailed overview of the particular classes (i.e. their attributes), their relations to each other and the multiplicities of each relation. Second, an explanatory statement is given to underline the reason why the particular class or relation is made the way it is in the data model.

Figure 3.4 shows the CVS data model in whole to give an overall view before we start examining the particular classes and relations.

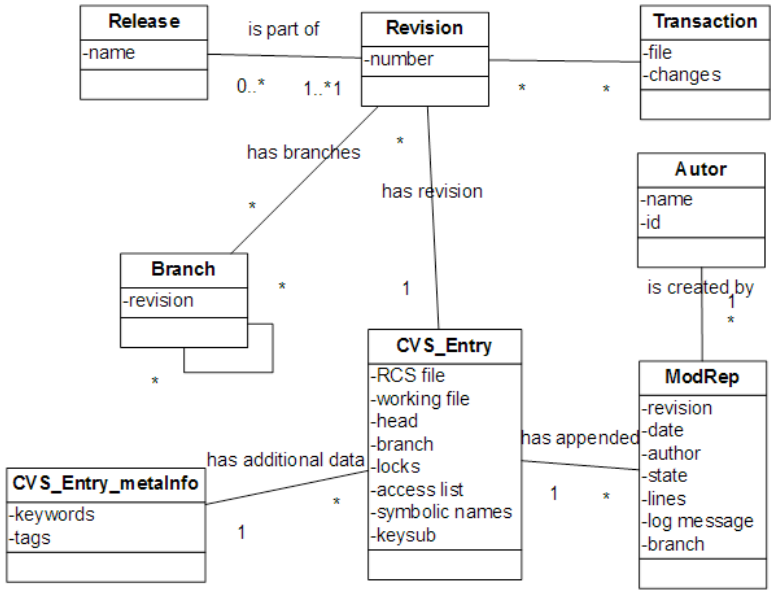


Figure 3.4: The CVS data model

The CVS-Entry - Revision relation

As seen in Figure 3.4 the revision is designated as a separate class despite the possibility to leave the revision information of a file as an attribute of the file entity. The CVS-Entry entity holds the following information:

- *RCS file*: The RCS file information is a reminder of the former Revision Control System (RCS). CVS still uses this format, particularly for history files, because the first program to store files in that format was a versioning system known as RCS. The RCS file data shows the path to the versioned RCS representation of a file in CVS (e.g. *\repository/directory/file.txt,v*). For detailed information about the RCS format and notations please see the CVS manual or the doc/RCSFILES file in the CVS source distribution.
- *working file*: The working file is the current name of a file the developer is manipulating in his working copy.
- *head*: Head represents the most recent revision of a file (the HEAD revision)
- *branch*: When a developer isolates changes onto a separate line of development, he usually creates a branch. The branch information shows all branches made at a particular revision. This information is however not always displayed at this place but rather in the modification report of a file (see the upcoming descriptions). If displayed, it shows only the branch point revision (e.g. if a branch is made at revision 1.2, the branch point revision would be 1.2.2; this means that a new branch file would have the revision set to 1.2.2.1)
- *locks*: Locks in CVS are meant to prevent complications in software development when multiple users change a single file. In an RCS manner, a lock is similar to a reserved checkout.
- *access list*: The permitted user list.
- *symbolic names*: Symbolic names refer to the a tag, a sort of file meta information. A symbolic name could be a vendor tag or a release or branch tag. (i.e. *1.5.2.1 : filebranch-2*)
- *keyword substitution*

The above data are considered relevant as they do appear in all the CVS file logs of different tools examined for this thesis.
The revision entity on the other hand just holds the information about the revision number of a file. As said previously, the revision information about a file could also have been placed into the file entity itself. The separation was done since a revision is a key information in the CVS data model.
The multiplicity of the relation is one - to - n. A file can have multiple revisions, where as for a file, a single revision is present only once (or a single revision belongs to one and only one file). Many files could indeed have the same revision; in this case the multiplicity would be n -to- m. The multiplicity in this relation is considered for the case of each file separately, not taking other files with possibly the same revision into account. From a modeling technique point of view it is more correct to look at a single entity and the data that comes with it than already linking more entities together. By doing so, a developer eventually faces some information loss when prematurely considering multiple entities of the same type in her basic model.

The Revision - Release relation

In both CVS and SVN, the release concept actually does not exist as a representation of an object or a data set. The release concept is among the central concepts in every (considered) release

history system so far. The more central role of the release concept is a reason why the CVS data model incorporates a separate release entity; which the revision is linked to.

The multiplicity of the relation between revision and release is in most cases n to m. In most cases because a revision does not always have to be in a release- in this case there would be no relation. In the Figure 3.4 the relation's multiplicity is 0..n to 1..m. This should simply show, that a release must consist of at least one file. In CVS even this does not have to be, because a developer can create a tag that is designated as a release but no files need be "attached" to the tag. Since this is a rather rare case in practice, it is not considered here.

The Revision - Branch relation

Occasionally, in larger software projects, the main line of development is split into several parallel lines, called branches. As described earlier, a branch has a specific number starting withe the two first comma separated numbers of the revision the branch is made of and additionally the the new branch number, starting with an even number.
The branch concept in CVS is, on one hand, tied to tags, since tags are used to designate a set of revisions to a release; on the other hand it is a part, or put more precisely, an extension to the revision concept, as branches are actually revisions themselves (revisions of revisions, but not to be considered as meta revisions).

Further, as Figure 3.4 shows, a Branch itself can have parallel development itself; a Branch can contain branches. The multiplicity of the Revision - Branch relation is one to n. A revision (a file, for that matter) can have multiple branches whereas a particular branch comes from a single revision. As for the CVS-Entry - Revision relation, the argumentation concerning the multiplicity is the same: the relation and its multiplicity are considered for one entity of each.

The CVS-Entry - ModRrep relation

A soon as changes to a file under version control occur, they are recorded in form of a modification report. In the case of CVS it is the appendix to a files revision log entry. Each modification report contains the new revision number, the date, the modification was made, the author and additional data[Ced05].

Another possible view is that the MR can also be coupled with the revision, since for every new revision there is a modification report. The MR is linked to the file, according to the model representation in the various tools considered for CVS.
The multiplicity is one- to- n. A File can have multiple MRs where as a particular MR belongs to one and only one file (revision).

The Author - ModRep relation

We have already noticed that a MR always has an Author that crated the change to a particular file and hence moved the revision up to a new number. The Author is considered to be a valuable information, thus it is placed in its own entity in the CVS data model. In almost every versioning system the Author plays a role, especially in change management systems, where the authors can be grouped into developer teams and can take different functions at the same time. For the CVS data model, the author has a unique name and an optional identification number.
The multiplicity of the relation is n -to- one; an author can be responsible for n modification reports, whereas a MR is written by a single author.

Transactions and CVS-entry-meta information

Transactions are not explicitly present in CVS. A transaction would designate a set of commits that leads to a new revision. This concept was added to the CVS data model, since a project[15] was concerned and implementing the transaction concept. For a more detailed description of a transaction, please see the elaborations in the SVN data model description in the upcoming section.
CVS-entry-meta information represents a set of additional information to a file, such as keywords or tags. This information is useful, since it helps in storing, finding and grouping files (tags) or it sets permissions or additional features to a file (keywords). The relation is made to the CVS-Entry entity and the multiplicity is one-to-n. A set of meta information is attached to one file, whereas a file can have multiple additional information attached to it (a file can be tagged for instance as a part of a release and it can be a branch- which again requires a tag of a different kind- as well).

3.6.2 The Subversion data model

The next release history system to be discussed is Subversion. We have already noticed, that Subversion (SVN) can be seen as a successor of CVS, incorporating improved versioning techniques in contrast to CVS.
The concept of versioning in SVN has been moved on to managing the history of entire directory trees instead of (only) managing particular files, as in CVS. So, each new revision is tied to a directory as well as to the file in that directory. On the following pages, the SVN data model (Figure 3.5) is being elaborated; in the same manner as the CVS data model, by first describing the entities and their relations and then giving a statement ,why the particular entities or relations are conceived the way they are.

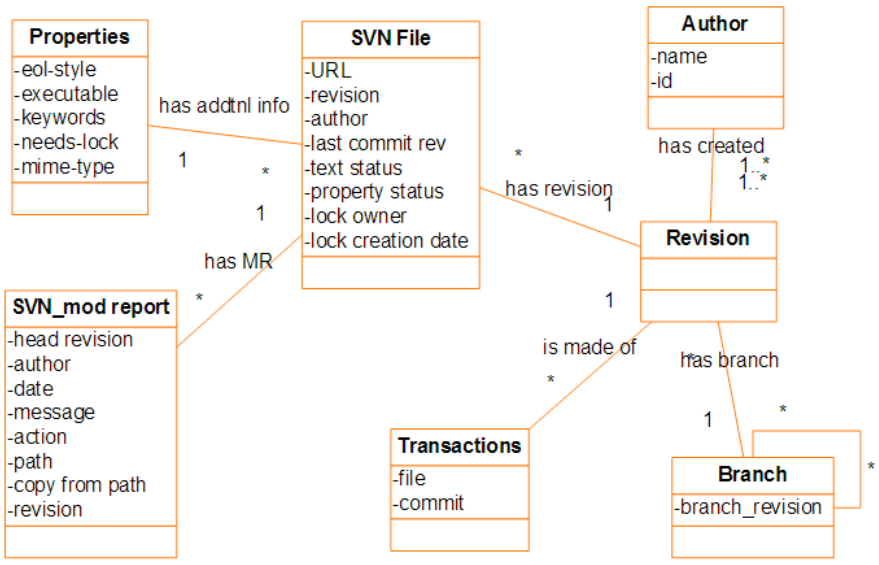


Figure 3.5: The Subversion data model

[15] s.e.a.l. research group devoted effort in explicitly recreating transactions

The SVN-File - Revision relation

In contrast to CVS, SVN is managing directories and thus files in them. Having in mind, that SVN actually does not make any distinction between files or directories, and the same information concerning release history applies to both the directory and file, we have taken the directory entity, the file entity respectively, into one UML class calling it the SVN entry . Further, we have designated the file and the directory as one entity, because the various tools, considered for SVN, make no distinction between a file or a directory either.

The SVN-File holds the following information:

- *URL:* The path to the file in a SVN repository.
- *revision:* the most current revision of a file or directory
- *author:* the author to whom the file "belongs"
- *last commit revision:* revision and the time stamp the revision was last committed
- *text status:* Tells whether a file has been modified either locally or both locally and in the repository, added or deleted.
- *property status:* Gives information about so called non versioned properties of a file or a transaction or directory tree respectively (i.e. a time-stamp, at which a transaction was created)
- *lock owner:* Gives the name of a person that made a lock (read, write) to the file
- *lock creation date:* holds the date and time, the lock has been applied to the file

The Revision entity holds the revision number without any additional information. The relation between the two entities is one -to- n; a file (directory) can have multiple revisions, whereby a particular revision belongs to a single file (again taking the single-entity-case as mentioned previously).

The Revision- Branch relation

As in CVS, a SVN entry can have multiple parallel lines of development (branches). When a developer creates a branch in SVN, a new file is being made, yet the branch file remains invisible for the developer. If she now wants to work on the branch file, a developer simply switches the current file to the branch file. What changes then, is the path of the file[16].
Internally, SVN makes a new sub-directory, when a developer creates a branch. What happens first, is, SVN creates a transaction tree, then after a commit the transaction tree becomes a revision tree with the new branch as a sub-folder or file. This procedure happens for all commit operations, no matter if a developer creates a branch or simply makes new changes to a main trunk.

Another similarity to CVS, concerning branches, is, that branches in SVN can have branches of their own. The multiplicity of the relation between the two entities is one to n; a revision can have multiple branches, whereas a branch belongs to a particular revision. The branch entity could have been also attached to the SVN entry entity, since a branch is a new file originating from a main trunk file, but conceptually, the revision is more important, since it is an unique and central information to a file (in the context of a revision or release history system), ergo, the branch entity is coupled to the revision entity.

[16]When switched to the branch file, the path to that branch file is being displayed when looking at the log

The Revision- Transaction relation

Unlike CVS, Subversion has a defined transaction concept. Transactions help in distinguishing a set of operations to a file that belong to a single development step as, for instance, a set of changes that lead to a new revision of a file (i.e. the current revision is 3; a set of changes is made that lead to revision 4).

A transaction in SVN represents a set of commits that apply to a file before the current revision changes to a new one (before the update command is being executed). The multiplicity of the relation is one -to- one; a revision is being made out of one set of commits, that is, a single transaction, which leads to a new revision, whereas a particular transaction (a set of commits) is bound to a single revision, meaning that each transition from one revision to another has a unique transaction behind it.

The Revision- Author relation

The reason, the author information is represented as a separate entity is the same as for the CVS model: it is a valuable information and a common concept in almost all release history systems. The author entity holds the name of the author and an optional id, if present. The multiplicity of the relation is n to one; a developer can be the author of more than just one revision, whereas a revision is made by one author.

The SVN-File- SVN-modReport relation

A modification report for a file in SVN can be extracted out of the history log for a file, since the log appears to be a listing of the different modifications for each revision of a file. Unlike CVS, where the modification reports are appended to the file log, SVN maintains the file and the modification information separately. So when looking at the modification report log, we see the particular actions (modified, added, deleted), the timestamps, the author, etc. for each revision.
The multiplicity of the relation is one to n. An SVN entry can have multiple revisions, thus, it must have multiple modification reports. A modification report (hereby, we will take an entire MR log) on the other hand, exists only once as such and is tied to one particular revision.

The SVN-File- Properties relation

Properties in SVN designate the additional information in form of tags or keywords to a file. For the sake of a later comparison, the properties are designated as a separate entity. The multiplicity of the relation is one-to-n. A SVN file can have multiple properties set, where a certain set of properties is attached to one file. The probably most interesting property are keywords. They are a common concept in many versioning systems (in this case CVS has the notion of keywords as well).

3.6.3 The Bugzilla data model

After detailed elaboration of the particular release history systems data models, the next step toward the conception of the meta model is to look in more detail at the data model underneath Bugzilla. The issue tracking data model is to extend the release history aspect of the meta model. Thus, we will examine the Bugzilla data model in more detail. We have mentioned in earlier sections, the the issue tracking systems are rather static in functionality, in comparison to release history systems. Nevertheless, the data model (Figure 3.6), is more grained than data models of

release history systems.

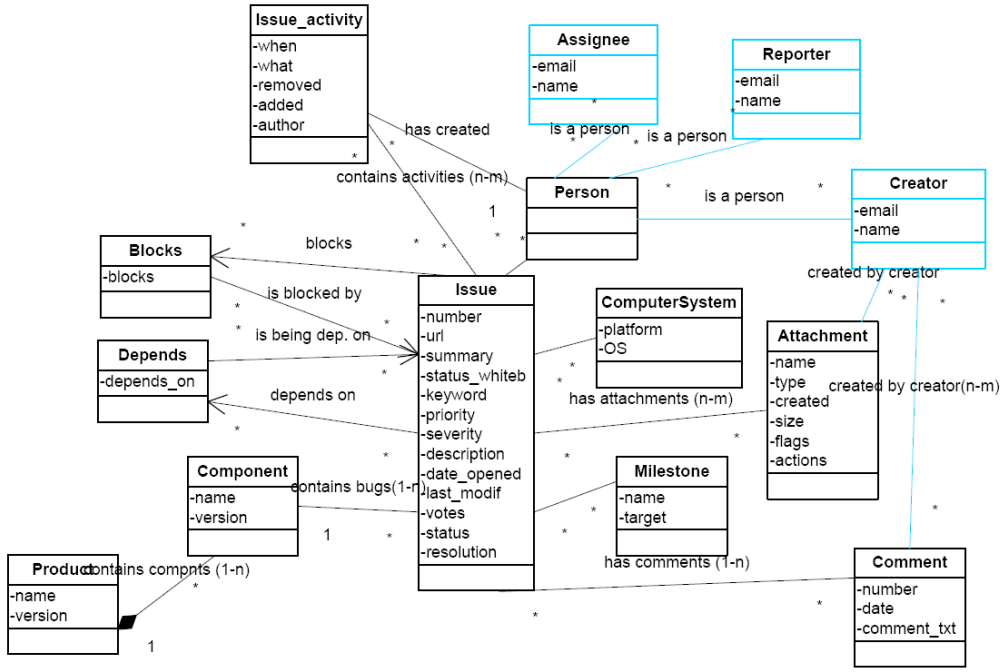


Figure 3.6: The Bugzilla data model

As it was done for the described versioning systems, the data model will be described in two ways. First, all the extracted entities, their relations and the multiplicity of these relations are discussed in detail, then an explanation will be given, why the particular relations and classes are designated as they are.

The Issue - Person relation

The central entity in an issue tracking system is of course the issue. The relevant data that comes within the issue entity is mostly standardized across issue tracking systems. In our particular model the issue entity holds only the information that is strictly related to it and can hardly be separated into a stand-alone entity. The issue entity holds the following information:

- *Issue number (ID)*: In Bugzilla, every bug has an unique number or ID. This number is increasing for each new bug submitted into Bugzilla. The bug number is an integer and increases by one for every new bug.
- *URL*: A URL associated with the bug. Usually pointing to a location, where additional information about the bug can be found. It is of course also possible to fill in a random URL unless restricted by the administrator responsible for a bug database.
- *Summary*: A short description of the bug.
- *Status white-board*: Somewhat similar to the Summary field, except that a larger amount of text can be written.

- *Keyword*: Used for bug categorization
- *Priority*: Used by the assignee (the responsible Developer for a bug) for prioritising the bug she is responsible for.
- *Severity*: As the name says, this attribute tels how severe a bug is.
- *Time stamp*:There are actually two timestamps: one, that tells the exact time, when a bug was submitted, the other, when a bug was last modified.
- *Status*: The status of a bug refers to the different states a bug goes trough in its life cycle [17]. The different states are reserved keywords such as "'new"', "'assigned"' or "'resolved"'.
- *Resolution*: If a bug enters the status of "'resolved"', there are multiple possible resolutions. Each resolution denotes a different outcome for the solution of a bug.

The person entity in this version of the model is a composition of three separate entities, that were present in earlier versions of the Bugzilla data model during this project.
To a bug, there are in essence two persons that are related to it. The reporter is the person who submits the bug to the data base. The assignee is the developer, responsible for the submitted bug. It is however possible that an assignee also submits a bug, which makes her a reporter. A bug always has a pair of persons that are related to it. Another person that contributes to a bug is a creator of a comment, an attachment or an activity. This person can also have the role of an assignee or a reporter or any of the just mentioned roles.
During the design of the Bugzilla data model, all the mentioned roles of a person contributing to a bug were designated as separate entities. It became clear, that all the different roles could be placed into one entity, the "Person" entity. Since all the different contributors to a bug are stored consistently by their e-mail address and optionally a name, the merging to a single entity presents no real problem of possible data loss or inaccuracy in information representation. Further, the relations of the entities[18] can be gotten rid of and we'd get a more clear data model. The former entities and their relations are painted blue in Figure 3.11.
The multiplicity of the relation among person and issue is m-to-n; an issue has multiple persons, contributing to it, where as a person can contribute to more than one bug.

The Person - Comment, Attachment, Activity relation

Having elaborated the person-entity problematics above, we will look at the particular relation of the person entity to comments, attachments and activities. Those three entities are considered to have the same relation for each, with the person entity so they are discussed in one turn.
The comment entity has a n-to-one relation with the person entity; a person can write more than one comment, where as a comment can be written by only one person at a time. This relation stands also for the activity and attachment entities.
In the implementation of the Bugzilla model, we will see that these relations are actually not one-to-n, but m-to-n, and they are implemented over association classes. Chapter 6 "Implementation and Evaluation" explains why that is. The model does not change however.

[17] See section 3.x "'Bugzilla"'
[18] The creator entity is linked to the comment, activity and attachment entities. Each of these entities is in turn linked to the issue entity.

The Issue- Comment relation

The comment entity holds information such as comment number, a time stamp, at which the document was created and the comment text. A comment is an additional information to a bug, mostly describing problematics that emerge during the solving of an issue. A issue can have one or more comments, whereas a particular comment is linked to a single issue. The multiplicity of the relation is thus one-to-n.

The Issue- Attachment relation

An attachment is also an additional information to a bug, with the difference, that an attachment is usually a file of any kind. An attachment has information such as the type of the attachment, date it was created, short description of the attached file and an attachment id. The relation to the issue is one-to-n. An issue can have more than one file- attachments, where as an unique attachment is linked to a single issue at a time. However, a file in an attachment can be, for instance a patch to not only one issue, but to many. Thus, the file would be attached to more than one issue and the relation would be m-to-n. But, the attachment is considered as an entity only with all its information. This means, the same file in an attachment can be attached to more than one issue, but it is likely that is was attached by a different person, at a different time, and perhaps with a slightly different description. Ergo, as a whole, the attachment is never the same for more than one issue, which confirms the one-to-m relation to the issue.

The Issue- Activity relation

As mentioned in the introduction to Bugzilla, in one of the earlier sections, every issue is strictly bound to a work flow in Bugzilla, called the bug life cycle. Due to this dynamic process an issue runs through, there is always a certain activity associated with an issue. The activity consists of, for instance, changes that happened to the status of an issue. Overall, the bug activity is a detailed record of all changes and contributions to an issue, including such tasks as comment-adding, status changes, etc. It is an important informational- content addition to an issue, that can be considered as a somewhat history of an issue. We have stated earlier however, that an issue does not have any history; what is meant, is rather the versioning aspect, that an issue does not have.
The multiplicity of the relation is one-to-n. An issue can have multiple activity instances[19], whereas an particular activity is linked to a single issue.

The Issue- Component relation

A component is a part of a product in the Bugzilla data model context. Issues are component- specific and can emerge in plural for one component. The component entity itself holds information such as the component name and the component version. The multiplicity of the relation is one-to-n. A component can have multiple issue, whereas a specific issue emerges in one particular component.

The Issue- Dependency relation

Earlier in the thesis, we have briefly stated, that issues are connected with each other by a certain relation. They can depend on or block one another. The dependency relation is however two-sided. To illustrate the dynamics of this relation, let's consider three bugs (B2, B4 and B5). B2

[19]Each activity-entry is considered a separate activity instance- not the set of activities for an issue.

depends on B4 and B5, B4 depends on B5. This is the ”depends” side of the relation. During the making of the data model, this relation is found to have a strict and direct opposite to its ”depends” side, namely the ”blocks” side. Strict and direct means, that when B2 depends on B4 and B5, B4 and B5 block B2. The same goes for B4: it depends on B5 so B5 blocks B4. It is not possible for B2 and B4 to block B5, nor is it possible for B5 to depend on either B4 or B2. The ”depends” side of the relation has for every issue a ”blocks” counterpart. The dependency relation is split into uni-directional relations to two entities -the blocks and the dependsOn entities- whereby, for either entity, two uni-directional relations are given.

The Component- Product relation

As mentioned above, a product is a set of components. Component entities have already been introduced. The product entity has a one-to-n composition-relation to the component entity, since we have stated that a product is a set of components. To illustrate this, see the following picture. Components and products are those entities that issues emerge from, and thus they are relevant enough to designate separate entities for components and products.

The Issue -Milestone,-Computer-system relation

Milestones are dates, that a developer plans to have her bugs fixed by. They can me set to certain dates or to certain development points (for instance, if somebody plans to have his bugs fixed for release 4.1, than the milestone is designated as release 4.1).
Milestones are designated as separate entities, because we wanted to keep the potion free, that if some more information is to be attached to a milestone, it can be modeled within the milestone entity separately, without having to manipulate other entities-that is, the issue entity, if the milestone were integrated into it as an attribute. A milestone entity has the name of the milestone as immanent information. However, a possible extension would be the date, the milestone is to be finished by, or the description of the milestone. The relation among issue and milestone is n-to-one. A milestone can be relevant for the completion of more than one issue, whereas an issue has to be dealt with up to a certain milestone. The remark here is, that the relation can also be n-to-m, since the Bugzilla documentation does not provide clear information, whether it is possible for an issue to be assigned more than one milestone. The idea for multiple milestones would be, that an issue has to reach a certain state in the life cycle up to a first milestone, and be definitely resolved by a second milestone.
The computer system is another entity, that was extracted out of the same reason as the milestone entity. So far, the computer system entity has the names of the platform and operating system, which a component with the issue runs on. Later, it is however possible, that these information can be extended with some other details. The relation is n-to-m. An issue can emerge in more than one computer system, and a computer system can have , although not desirable, multiple issues.

By now, the overview of the different versioning and bug tracking systems is complete. An introduction and a classification was made, and the relevant systems were described and their data models elaborated. This chapter serves thus, as a pre-meta modeling milestone, from where the release history meta model is being developed.

Chapter 4

Developing the release history meta model

This chapter describes and discusses the making and details of the release history meta model and its extension with the issue tracking data model.
We have introduced and discussed release history and bug reporting systems that are going to be the conceptual base for this meta model. We have taken the particular systems, and derived the respective data models in an object oriented fashion into UML diagrams. The next step is to build a meta model, that is capable of incorporating all immanent concepts and the most important and relevant information of the different versioning systems considered in this thesis.
We will start with an introduction to meta modeling. The idea is to provide a short theoretical base, to what meta modeling is, and how the modeling technique is applied in this thesis. After this short theoretical introduction, we will start developing our meta model. The procedure is split into two major parts. In a first step, the meta model is derived from the CVS and SVN data models and extended with the Bugzilla issue tracking data model. In the next step, the release history aspect of the meta model is considered again and modified. The modification concerns the file or Entry entity of the model, whereby the modification is about splitting the file entity in more sophisticated entities. This modification is done with particular focus on source code projects, where a file or entry can be further split into packages, classes, methods, etc.

4.1 Modeling concerns for the release history meta model

The problematics of information abstraction have a central place in the context of designing the meta model in this thesis. Namely, the problematics of our work can be put in relation with the facts, mentioned in Chapter 2 under "Meta modeling" up to a certain degree. As a developer, who deals with different concepts and abstraction levels in designing a software system, or a developer describing the grammar of languages in order to classify them, we face a problem in describing and combining different concepts that are present in different systems of one type, namely release history systems. Each of these systems has its own conceptual world and each of them has a sort of a different abstraction level immanent to it. However, the abstraction levels are not so different, so the meta modeling had not been too concerned with finding a suitable abstraction level for the meta model. By using a meta model, in the UML2.0 conform notation, we have been able to extract the similarities of the particular data models and describe them consistently in the release history meta model.

In the approach to meta modeling in this thesis, we have conducted a somewhat different process of constructing a meta model, than it would be, harshly said, commonly known.

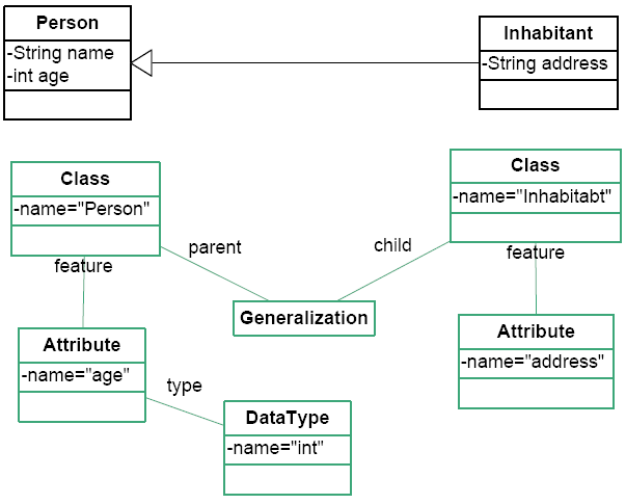


Figure 4.1: Different meta modeling approach

In Figure 4.1, a simple relation between two UML classes is described in abstract syntax as instances of a meta model. Hereby, each information to a class (name, attributes, etc.) is classified and instantiated.The under part of the picture holds the meta data and is used to describe relations of the shown sort. The meta modeling approach for this example derives a different abstraction level but remains in the same conceptual world (the Person-Inhabitant -class generalization) [GT05].
Our concern in this thesis is a slightly different one. We do not burden ourselves with describing one model with meta data, but rather, conceptually examining different data models and, based on similarities, deriving a meta model, that describes all the data models. An appropriate example, for the sake of comparison to the approach in the above shown picture would be as follows.

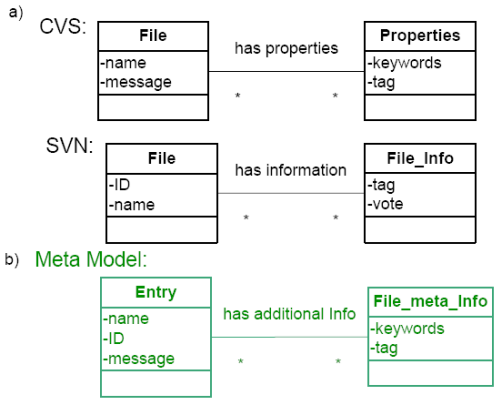


Figure 4.2: Used meta modeling approach

In Figure 4.2 under a), the CVS and SVN relations among a file and its additional information are shown. The Abstraction level in these two concepts are similar, what differs are the type the

relations and the attributes in each concept. Our approach is, first, to recognize the two classes as common in each of the concepts, then designate them as entities in the meta model with the relevant information. Further the relation and its multiplicity in each of the concepts is recognized as common, then designated as a relation between the entities in the meta model. The semantics for each data model (CVS designates the shown relation as ”has properties”, SVN as ”has information”, multiplicity of the relation, attributes to a class) are not changed significantly and the abstraction level is slightly changed (A file is designated as an ”Entry”, etc.), but what we derive in b) is a class and relation description, that can be valid for not just CVS or SVN, but for every other release history system.
The kind of meta modeling we use in this thesis is focused on similarities between instances of different conceptual worlds as in the similarities in the semantics of the different concepts. The similarity problematics are more pertinent, since the abstraction levels and the semantics of the different release history data models are mostly the same.

4.2 Deriving the meta model

During the definition of the entities and their relations in the CVS and SVN data models we have discovered some similarities, thus common concepts among release history systems as well as some tool or system specific differences. Having these relations among the two data models in mind we extract the core entities, that represent the main concepts of release history systems out of these models and start building the meta model.

4.2.1 The Entity-Revision relation

The Entity, taken as a representation of a file until its further specialization in upcoming sections, has both in CVS and SVN a revision attribute. In the model, we extracted this attribute and designated it as a separate entity (Figure 4.3). Two main reasons for this are pointed out: first, the separate Revision entity makes it easier to map other entities to it and to the model; second, every release history system has the revision as a core element of the particular versioning concept so at least here it is clear, why the Revision entity is rather important.

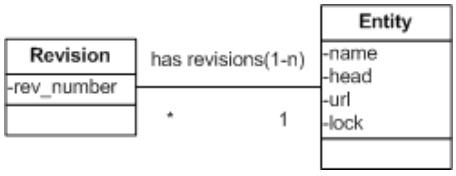


Figure 4.3: The meta model file- revision relation

To illustrate the revision as an important information consider the log entries of CVS and SVN. The next listing shows an CVS log as taken from the WinCVS[1] tool

Rcs file : ’”ïestrepo/someth/test.txt,v’
Working file : ’test.txt’
Head revision : 1.8
Branch revision :
Locks : strict

[1] www.wincvs.org

Access :
Symbolic names :
1.5.0.2 : 'rel-1-5'
..

————————————-
Revision : 1.8
Date : 2005/8/24 13:39:58
Author : 'mobsys'
State : 'Exp'
Lines : +0 -15
Keyword : 'kv'
CommitID : 'df8430c78ad7112'
Filename : 'test.txt'
Description :
no message
————————————-
Revision : 1.7
Date : 2005/8/22 15:58:25
Author : 'mobsys'
State : 'Exp'
Lines : +4 0
Keyword : 'kv'
CommitID : 'e844309f6203ec9'
MergePoint : '1.5.2.1'
Filename : 'test.txt'
Description :
no message

The example for an SVN file log on the other hand is shown as in Figure 4.4, where the revision is explicitly designated.

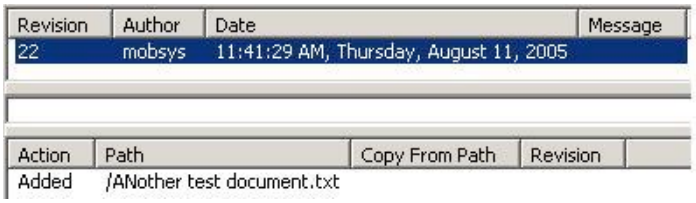


Figure 4.4: Subversion file representation

The next important facts are the relation and the multiplicity of it among the entities: As seen in Figure 4.3 there is a n-to-1 relationship between Entity and Revision. This means that an Entity (file) can have up to n revisions where as a revision is clearly assigned to one file. The CVS log excerpt shown above illustrates this relation. Revision 1.8 is the head revision of an Entity (here, a CVS file). Revision 1.7 is the next older revision of the same CVS file. This denotes, that a file has multiple revisions stored in its log.

For SVN on the log entry for a file and its revisions looks as in Figure 4.5. Revision 17 is the head revision, and revisions 16, 15 and so on are the next older revisions respectively.

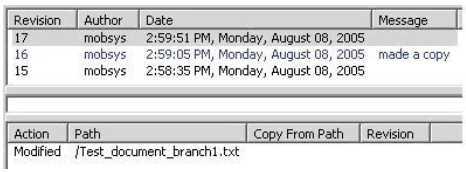


Figure 4.5: Subversion log entry

4.2.2 The Revision-Author relation

A file entity in each versioning concept has the Author as an attribute. The author plays an important role in the versioning concepts since we wish to track down changes to a certain Author at a given point in time.
So what we did was to again separate an attribute, in this case the Author- attribute and make it a separate entity in our meta model as well as in the data models of the particular versioning systems used as a base for constructing the meta model.

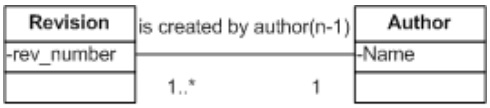


Figure 4.6: Revision Author relation

The attribute for each entity here is basically clear. A Revision is defined by the revision number for a file; an Author is defined by his or her name (Figure 4.6).
Taken that a person is considered an Author when he or she is logged into the system or has his or her own working copy of a file or set of files, the relation and the multiplicity defining it can be put to like this: A revision has one author that made that revision. This statement stands before the implication that a new revision is created after a commit command which incorporates several changes a single author has made to a certain file. So, for a revision there is one and only one author.
The other end of the relation states that an Author can have her name on one or more revisions; again considering an author to be a logged in user or a user recognized by the release history system.
To illustrate the relationship see the again Figure 4.5; the second row denotes one author for each revision.

4.2.3 The Revision-Transaction relation

This relation is not explicitly given for all versioning systems taken into account for this thesis[2]. In SVN for instance there's a transaction concept that clearly shows a transaction as a sum of several commits made by an author. After each sum of commits in SVN a new transaction tree is being generated out of the versioned folders tree. Then a new revision of that tree is being stored in the SVN database.
The concept of transactions however remains the same across the two versioning systems.

[2]The CVS versioning system has no explicit notion of transactions as described in t section 3.6.2 "The SVN data model". The transactions had to be reconstructed. For more detail, please see [FPG03].

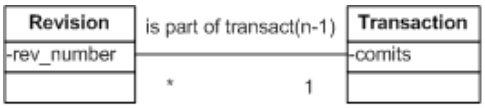


Figure 4.7: The revision transaction relation

The multiplicity of the relation describes the following relation: A revision has one transaction that created it whereas a transaction can influence more than one revision (depending on how many files with different revision numbers are being affected by the commit) (Figure 4.7).

4.2.4 The Revision-Release relation

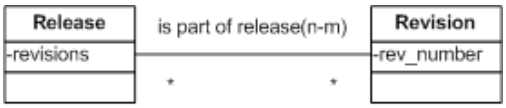
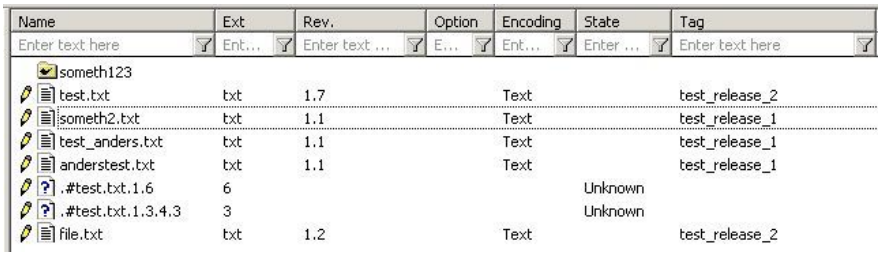


Figure 4.8: The release revision relation

The Release entity is firmly coupled with the concept of file meta data such as tags, labels or keywords. The reason is that there is actually no explicit release- entity as such. The release concept is however very helpful in grouping files and distinguishing between different programming lines of a piece of software.
A release is generated trough adding meta information to files that designate the file to a particular release. In order to more clearly depict the release concept that is immanent to most release history systems including the ones considered in this thesis we have taken the release as a separate entity that is linked with the Revision entity - since a release consists of one or more revisions. The multiplicity has been partially explained (from the release point of view). On the revision side it's clear that a single revision can emerge as part of several releases. Figure 4.8 designates the relation.
Figure 4.9 illustrates the relationship in an excerpt of the WinCVS log. The file information contains additional information ("test-release-1") that designates it to that specific release.



Name	Ext	Rev.	Option	Encoding	State	Tag
Enter text here	Ent...	Enter text ...	E...	Ent...	Enter ...	Enter text here
someth123						
test.txt	txt	1.7		Text		test_release_2
someth2.txt	txt	1.1		Text		test_release_1
test_anders.txt	txt	1.1		Text		test_release_1
anderstest.txt	txt	1.1		Text		test_release_1
.#test.txt.1.6	6				Unknown	
.#test.txt.1.3.4.3	3				Unknown	
file.txt	txt	1.2		Text		test_release_2

Figure 4.9: CVS release information

The above figure illustrates the relationship for CVS. In SVN the concept is the same, though there is no distinction between a branch and a tag, meaning a release is no different than a branch of a folder structure versioned under SVN except that when you separate a part of the folder structure as a release you must not change any files in that release because it would then turn into

a common branch. The relation between the release and a revision of files and the multiplicity of the relation remain the same in SVN as in CVS.

4.2.5 The Revision-Branch relation

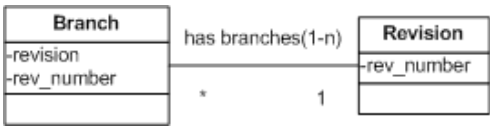


Figure 4.10: The revision branch relation

When working on a file there is always the possibility that this file can be used (slightly modified) for some other purpose than the one you designated it for (by another person, another department in your company, etc.). In this case you separate a branch of that file (or a set of files) and treat it as a separate development line. This is a common concept in versioning systems. The relation in Figure 4.10, between a Revision and a Branch has an n-to-1 multiplicity, which means a Revision can have multiple Branches whereas a Branch comes from a single file, ergo, from a single Revision. A Branch can itself have its own sub branch. To illustrate the relationship, consider Figures 4.11 and 4.12. Figure 4.11 shows the excerpt of a SVN branch tree in form of a graph, where branches originate from one revision or from other branches. Figure 4.12 shows a CVS revision tree, where the branch point is at revision 1.3.

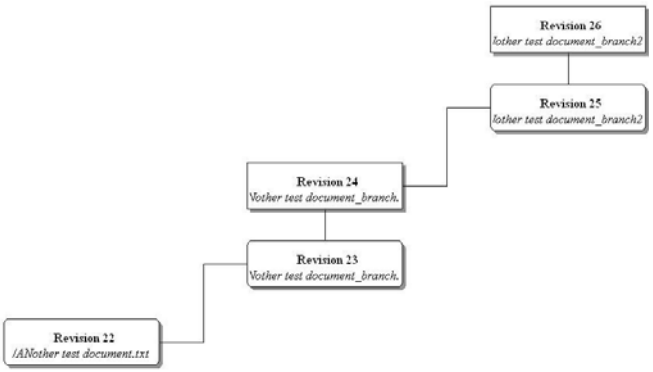


Figure 4.11: A branch graph in SVN

4.2.6 The Revision - Modification Report (MR) relation

Each time an author changes a file and commits the changes, a new revision of that file is being made. In relation to this a modification report is being generated telling a third user who, when, on what file has made the change. From the versioning tool's point of view the modification report (MR) is linked to a file instance in the versioned repository. But from a conceptual point of view it would be more correct to make the relationship between revision and MR. The reason is the taught that each MR describes a new revision thus it's in direct relation with a revision; which is also indirectly linked with the file instance (see complete meta model picture for details).
The relation is described as follows: For a single revision there is one MR that holds the data about

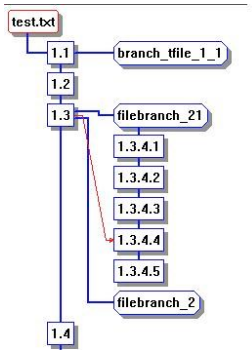


Figure 4.12: A revision tree in CVS with branches

the changes that led to that revision. A MR is linked to a single revision, because it describes exactly the one revision it is linked to (Figure 4.13).

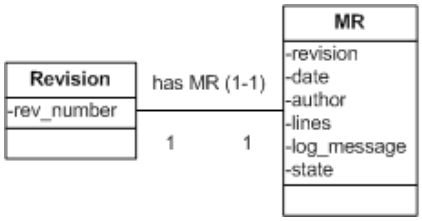


Figure 4.13: The revision modification report relation

4.2.7 The Entity - file-meta-info relation

By examining the different versioning tools we have discovered that a file instance has a set of ”meta” information attached to it and describing the file. We designated the meta information as a separate entity. What follows is a 1-to-n relation with the Entity instance (Figure 4.14). Ergo, an Entity (or file) can have one more of these meta information whereas a single meta information instance is linked to exactly one Entity, namely that on that it is describing.

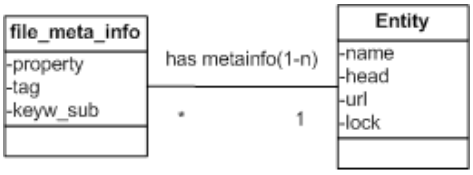


Figure 4.14: The Entity - file-meta-info relation

4.3 Extension of the meta model with the issue tracking data model

In the earlier discussion of issue tracking systems in this thesis, it was stated, that the issue tracking domain can be considered to be a sort of extension to the release history systems. It was said, that issues can emerge from one version of a file to another and thus issues are closely related to versioning. The issue tracking domain is however more than an addition to release history problematics. It is a sophisticated system for bug and change/feature request management. In the case of the release history meta model, we develop in this thesis, the issue tracking domain is indeed an extension of the release history data model. By having a sophisticated release history meta model combined with issue tracking data, we get a more complete base to model the entire life cycle of a file under version control. By adding issue tracking information to a release history model, the interactions among files are further extended with information in form of problem reports that again have their own particular structure. The result are two inter-operable models with the purpose of history and problem management of files.
For the purpose of combining the release history meta model with issue tracking data, the Bugzilla data model was introduced and discussed.

4.3.1 Linking release history data with issue tracking information

When editing a file under version control, a new revision of that file is made along with a modification report for that revision. When looking into modification reports of files under version control in more detail, i.e. into commit messages, which textually describe the changes made and some additional information, certain patterns can be found that might be useful in construction the bespoken link between versioning and issue tracking. Namely, commit messages often hold keywords, such as ”BUG”, ”bugID”, ”bugNum 2315” or similar, that point to an issue that emerged during the creation of a version (modification report), the commit message is appended to. These keywords indicate, that a modification report presents the linking possibility with an issue tracking model. While examining the release history data models, commit messages came up to be the only considerable linking solution to issue tracking data. First, modification reports are, as described in earlier sections, closely related to a file and thus to the revision of that file-hence, if we'd to look in a transitive way, the relevant link to a revision, as a central entity in the release history meta model, is present. Further, the modification reports are the only entities that hold the kind of information, that is relevant for the linkage.
On the side of the issue tracking data model, the linking point would be the issue entity directly[3]. The data relevant for linking would be the issue ID, since modification reports refer in their commit messages to such issue numbers or IDs. The following figure shows the excerpt entities from the release history meta model and the Bugzilla data model with the established linking, bidirectional, many-to-many association.

To come back to the introduction to meta modeling, where meta modeling is considered to be the effort on combining different conceptual worlds and describing their semantics, the application of this approach is hereby shown again. There are two conceptual worlds: the release history concepts and the issue tracking concept (Figure 4.15). In order to enable a comparison and combination they are modeled with the same modeling language (UML) and in consistent modeling style. The static semantics (the structure), as well as the dynamic semantics[4] of the concepts (the data models) are different[5]. The abstraction levels differ. Entities in the release history

[3] See section 3.6.3 ”Bugzilla data model”, Figure 3.6
[4] The behavior of the two systems, meaning the interactions and dependencies among entities
[5] If we abstract from the notation style, which is the same for both concepts in this particular case, the release history

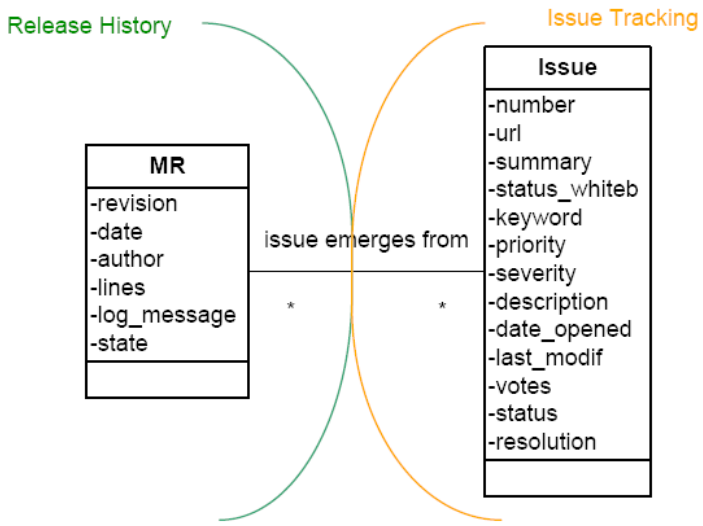


Figure 4.15: Bidirectional linking association of modification report and issue entity

concept designate data on a source entry (file) level, whereby the issue tracking concept abstracts information on a file-data level -the designated data is about an information to a source entry. Nevertheless, the two concepts can be combined, interfering neither semantics nor abstraction level of the particular concept.
The multiplicity of the linking association is many-to-many. A modification report can reference to one or more issues, whereby an issue can emerge in more than one modification report.

4.4 Further specialization of the release history aspect of the combined meta model

The issue tracking aspect of the combined meta model abstracts data on a level that allows the linkage not only on a file level, since the information in it is abstracted on a data-to-a-entry level. This is an interesting fact to remember in the following effort to further specialize, or if one wants so, to change the abstraction level of the release history meta model. The reason why this effort is made, becomes clear when the focus falls onto the source code development domain in software engineering. Release history systems usually are capable of managing pure text files as well as binary files. Under text files we commonly understand either plain, unstructured text files or source code files of some particular programming language such as Java, C++, HTML, etc. The latter sort of text files will be of further concern in this section.
Release history systems manage entire source code files without or with little notion of the text structure in the file. This means, that a release history system, such as SVN, captures the changes made to a file, thus crates revisions and point to conflicts, when merging conflicted code from a branch, but it has no notion of what exactly has changed in the file while changes were applied to it. It does not recognize, whether a method declaration was altered or an attribute was changed in a source file. However, there are certain versioning systems that do consider the internal structure

meta model and the issue tracking model show different structural details.

of a source file[6], and manage not only files, but more detailed entities, such as classes, methods, attributes, etc.
So far, the release history concept's abstraction level was on a file- level, not considering the internal structure or the type of a file. But, due to development on fine grained versioning systems and augmented focus on source files, the revision history meta model is to be further specialized in its entities and thus its abstraction level.

The idea is to further lower the abstraction level in a systematic way, to be able to encompass, on one hand the different types of entries, a versioning system can manage (files, directories), and on the other, to emphasize the internal structure of a file.
The specialization of the file entity is done according to Figure 4.16.

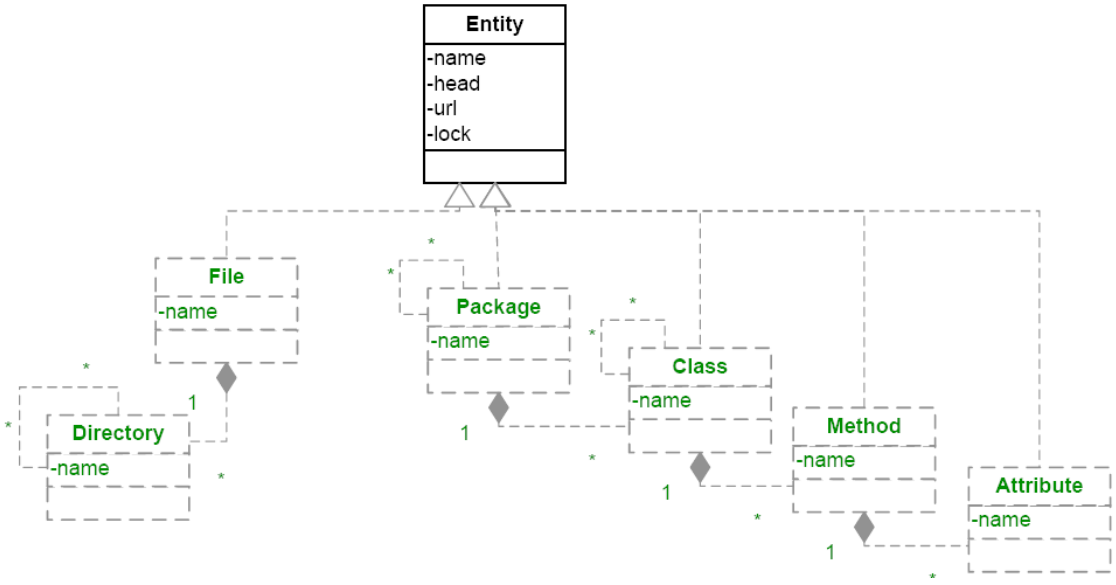


Figure 4.16: Specialization of the release history meta models file entity

Instead of a File entity as the central link to Modification Report, Revision, etc., there is now the "Entity" instance. The Entity has two ways of specialization. In one way, the specialization is made on a file level, distinguishing between two sorts of objects a versioning system can manage, namely, a File, and as a composition related to the File, a Directory, whereby a Directory can have one or more subdirectories[7]. The other side of the specialization is concerned with the internal structure of a managed object, hereby having the focus on source code files. The Entity is an abstraction of a Package -the Package being the hierarchically highest object in a source code domain[8]. A Package can in turn have one or more sub-packages. A package is further a composition of Classes, which again can have subclasses of their own. Further abstracting, a Class can have Methods, which in turn can have none or many attributes. Attributes designate the lowest abstraction level and the lowest hierarchical stair for this effort of specialization.
The accomplished change in the abstraction level of the release history meta model enables the

[6]One such fine grained release history system is being developed at the University of Salerno, Italy, under the direction of Prof. Andrea de Lucia
[7]With modeling directories as shown, it is possible to organize a hierarchical structure of the managed Objects, if such a necessity arises
[8]The Java programming language is taken as reference, while specializing the entities

modeling of a wider spectrum of versioning systems, hereby encompassing the fine grained versioning systems, that manage internal structures of files as well, yet still being able to efficiently model common versioning systems such as the described systems of SVN or CVS.

4.5 The combined meta model overview

After the completion of extension and specialization of the release history meta model, the graphical interpretation looks as in Figure 4.17.
The upper right entities in the Bugzilla model part (Assignee, Reporter, Creator) are all merged into the Person entity. The respective relations of these Entities are to be considered as relations of the Person entity.

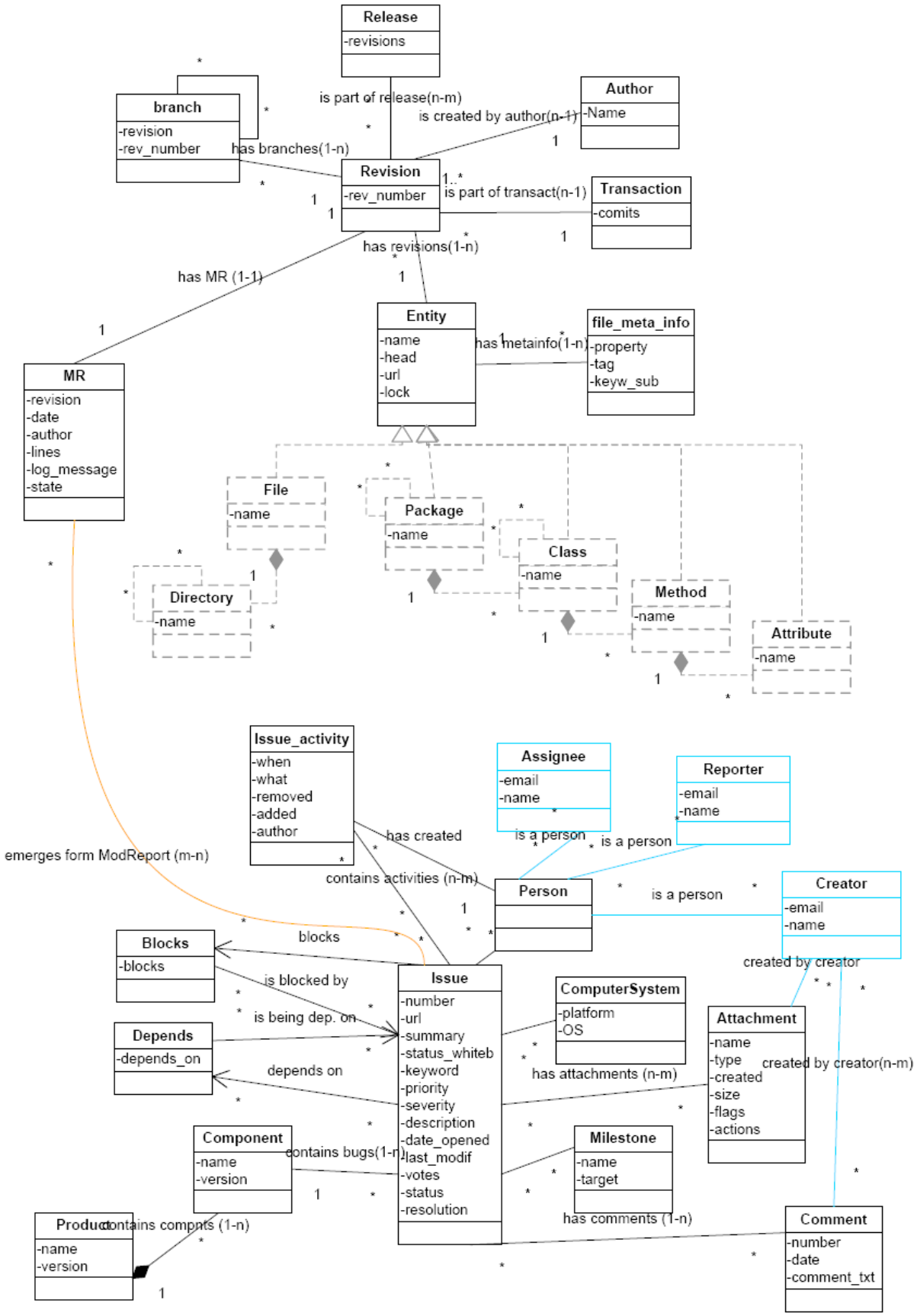


Figure 4.17: The complete extended and specialized release history meta model

Chapter 5

Validation with ClearCase

While examining the CVS and Subversion data models all the necessary tools and documentation were relatively easily accessible and free to use. This is because the CVS as well as the SVN systems are open source projects. The documentation provided was insightful and detailed.
With Clear Case it was a slightly different story. First, and maybe most important, is the fact that Rational's ClearCase is not an open source System, ergo, there is no free or trial version of the system. The next point is the ClearCase documentation. Though there is plenty of free and downloadable documentation, the documentation is not quite detailed about the actual data model of ClearCase nor is it consistent in its definitions of certain aspects of ClearCase. However, the documentation provided a solid base for a more detailed data model.

The release history meta model was derived on the base of release history systems, that, according to the implicit classification of versioning systems earlier in this thesis, fall under the first class of verisoning systems, that is, they are pure versioning systems. These systems are sophisticated and incorporate the common and most significant concepts in release history management. However, in order to underline the relevance and a wide applicability of the meta model, it has to be validated against a release history system, that was not part of the informational base for its construction. For the purpose of validation, Rationals ClearCase release history system was used. The validation is done by, first, identifying the ClearCase data model in a fashion, all the other data models were derived, then the ClearCase data model is compared with the release history meta model. During the comparison, the focus lied on identifying significant similarities among entities and particularly their relations. These similarities refer to common versioning concepts such as revision, release, modification report and the relations among these concepts. It is to be said, that the validation is not concerned, as to how, for instance, ClearCase stores versions (revisions), but rather, whether the release concept is present in ClearCase and whether the concept can be modeled with the semantics of our meta model.

The validation in this section will start by introducing the ClearCase data model. The derivation of the particular entities and relations of the ClearCase data model is not the primary concern, thus it will not be explicitly emphasized. The entities and relations will be explained in the context of the entire ClearCase data model and during comparison of the two models. The final section of this chapter will then emphasize some interesting ClearCase concepts related to versioning, that would extend the release history meta model in order to further span its relevance for change or configuration management systems.

5.1 The ClearCase data model

While extracting the relevant information for the data model in the ClearCase documentation, some details regarding information to entities such as attributes or the exact multiplicity of the particular relations, are inconsistently explained or not present. However, the ClearCase data model, as it is shown in Figure 5.1, is valid enough for the effort of validation in the upcoming section.

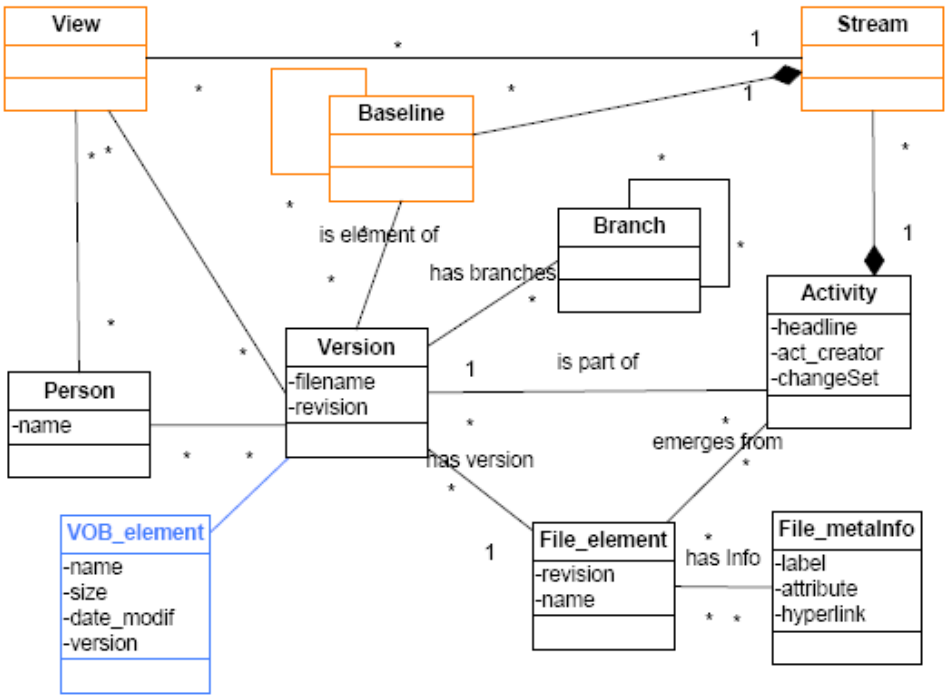


Figure 5.1: The ClearCase data model

The under right entity of a VOB[1] element is present, and its attributes are in place, yet it remains unclear, due to the ClearCase documentation, what multiplicity the relation with the Version entity has. The upper entities represent the concepts in CleasCase, that will be discussed in context of extending release history systems in one of the next sections.

5.2 Validation

As mentioned earlier, the validation process is focused on finding similar concepts (in form of entities) and relations in both models. The elaborations will be done by first taking a ClearCase entity and relating it to the appropriate entity in the meta model. During validation, it will be possible for some of the entities and relations to be compared in more than one way. These possibilities will be mentioned, since some concepts in CleasCase can be considered ambiguous as

[1] Versioned Objects Base is a sort of repository in CleasCase

they can act as representations of more than one concept[2].
Before starting the actual validation, the parameters of a successful validation have to be pointed out. Since the release history domain is of concern here, there are core concepts in release history, that need to be covered in a satisfactory way, in order to successfully validate. These concepts are the following:

- File/Directory
- Revision/Version
- Branch
- Release
- Author/Editor
- Modification Report
- Additional file information/File-metainfo

If these concepts, that are present in ClearCase, can be modeled by the semantics of the meta model, than the parameters of a successful validation are satisfied and the meta model is valid for more systems than the ones used for its construction.

The File/Entry validation

A File in ClearCase can be either a text file (source code) or any sort of a binary file. The file as an entity is present in both models, which was straightforward, since files are the most common managed objects by a versioning system. Further, ClearCase manages directories as well. The directory entity is not designated explicitly, since there is no reliable information, whether it can be considered as a specialization of a file or as a separate entity, thus the relation between the two models is made among the file entities, which is satisfying as well. The meta model's semantics are able to model the ClearCase file concept.

The Revision/Version validation

Each file in ClearCase has a version. Hereby, newer versions of a file can be stored as separate files, without changing an older one. The same stands for directories. The version of a file in ClearCase was designated as a separate entity. As the picture shows, the revision entity in the meta model is unambiguously the counterpart to the ClearCase entity.

The Branch validation

Another concept, common to most versioning systems is branching. A branch in ClearCase represents a parallel development line as it does appear in other versioning systems. The branch concept is also validated, however, ClearCase manages branches in a proprietary way. Namely, most versioning systems don't have a notion of a branch until a developer really branches off a separate development line. ClearCase however, defines every development line as a branch, whereby the initial development line is designated as the main branch. From this main branch, all other branches are derived. This idiosyncratic feature of ClearCase makes however no difference to the overall notion of a branch.

[2] An Activity in CleasCase, for instance, can be a considered as a transaction set on one file as well as a change set, thus a directory, dependent the desired role of the Activity, that is, on the information from it that is required

The Release validation

The release concept poses some deviations in the straightforward validation. A release is a set of revisions of multiple files. All the revisions in a release are tagged with additional information, that designates them to particular releases.
In ClearCase, the release concept could not be identified as such. It is however present, under different names and functionalities. Two entities in the ClearCase data model could be identified as a counterpart to the release entity in the meta model: the Baseline and the Activity entity.
A baseline, according to [Rat03a], identifies one version of each file element in a component[3] that represents the work of a group of developers. It designates a version of a component at a certain stage in development. This definition of a baseline corresponds to the notion of a release in other versioning systems, such as CVS or SVN. A release is a set of files, at a certain revision, that designate a software component or the result of a certain stage in development effort. Thus, the base line is a possible candidate for the validation purpose.
Another concept in ClearCase, that might as well be the counterpart to a release is the Activity. An Activity is a ClearCase object, that encompasses a set of files, that a developer creates or modifies. This set of files is the change set of an Activity, and it represents development tasks, such as bug fixes or, even releases of software components. The difference between a baseline and an Activity is in the number of developers, that are appointed to either of the two. A baseline has many developers, whereas an Activity exists for one developer only. Nevertheless, the Activity can be a set of versions, that designate a release in its common notion, ergo, the Activity is the second possible entity to be validated against the release entity of the meta model.
Whether baseline or activity, as long as the meta model's semantics incorporate the release concept, it can model both baselines and activities as releases. The validation for the release concept is accomplished, however with a certain ambiguity, whether baseline or activity should be considered as a counterpart to a release. From the point of view of the author of this thesis, the baseline is considered for validation, since the activity is more ambiguous in its definition of a set of files it incorporates.

The Author/Editor validation

ClearCase manages developers as individual contributors or as groups (teams). Each developer has an account and a working copy (view) of a set of files. The Author entity models developers, both as individuals and teams, since the information to an author consists mostly of a name and/or an ID. The relation between Version and Author in the ClearCase data model is m-to-n because of the just mentioned consideration. The meta model considers an Author as single instance -it abstracts the group notion of an Author as a single instance[4]. Thus the relation between author and revision has the multiplicity of one-to-n. The Author concept is valid, if an author is considered as a single instance. If the multiplicity is changed to m-to-n, the meta model could model groups as well[5].

The Modification Report validation

The modification report validation faces similar problematics as the validation of the Release entity. It was said earlier, that ClearCase stores each new version of a file as a separate object. These objects are stored in the VOB and are referred to as VOB-elements. During the data extraction for the ClearCase data model, no direct counterpart to the modification report entity was found. The

[3] A component is considered to be the object of a development effort; a piece of software, a module, etc.
[4] The versioning systems used for constructing the meta model do not have the notion of the author as a group. However, it is possible to extend the meta model in that it models the author as a group as well
[5] Please consider the upcoming section for the proper extension of the author concept

VOB-element entity, however, turned out to represent a similar concept to a modification report. Namely, each time a new version of a file is created, a new VOB-element is stored in the repository. This VOB-element holds the information about modifications made to the file (time stamp, size, author of the modification, etc.). A VOB-element is thus considered to be a modification report in form of a stored physical object in a repository. The meta model allows the handling of modification reports as they are present in ClearCase.

The Additional-file-data/File-metainfovalidation

The file-meta information concept is present in ClearCase in the same notion as it was designated in the meta model. The information, the file meta info entity in ClearCase holds attributes such as labels, attributes, hyperlinks, etc. A label, for instance, designates a user-defined name for a version, which corresponds to the tag attribute in the meta model.

5.3 ClearCase features as possible extension to the release history meta model

The validation against ClearCase is accomplished within a satisfactory spectrum. The core concepts and relations of release history systems could be validated, though, some ambiguities remain concerning revisions, authors, and modification reports. Ultimately, they do not relativize the modeling of the ClearCase data model with the semantics of the release history meta model. We look at certain ClearCase specific features that can be considered for an eventual extension of the meta model's semantics. These features can be implemented in the meta model since they seem very plausible and applicable in release history management as well as from a conceptual point of view.

5.3.1 The "View" Concept

A view in ClearCase is a similar concept as a working directory in other versioning systems. However, the working directory is not a concept imanent to a data model of a versioning system, but rather an implementation concern. The view concept in ClearCase abstracts the view as a part of the data model. It defines a set of files in a specific version or just a set of files, accessible to a developer. This concept enables a better overview of efforts done by many different developers or teams. Integrating this into the data model allows a further layer of organization of files or directories.

5.3.2 The "Activity" Concept

The activity can also be considered as a change set, a set of files, a developer currently works on. An interesting addition from ClearCase would be the additional information, that come with a change set; information related to the author. Namely, so far change sets can be modeled with the meta model as directories (a set of files) with indirect relation to revision and author . The Activity would be a separate entity, encompassing a set of files with additional information to the author who is assigned to it (and to the revision of course) directly.

5.3.3 The "Stream" Concept

A stream is probably the most interesting feature, that can be added to the meta model. A stream is a ClearCase object[6], that maintains a record of activities and baselines. Further it determines, which files are shown in a view for a developer. It is considered to be the work flow history managing entity in the ClearCase data model. This would be a completely novel concept in the release history meta model. A stream entity in the meta model, would be the composition of modification reports and it would have additional information about files, authors, and activities.

[6] An object in the sense of a Stream is the information representation towards the user. A Stream is the working environment with all the visible change sets for a user. It encapsulates a development effort or task in ClearCase. A Stream is not an object, but rather the set of concepts, such as Activity or Baseline.

Chapter 6

Implementation and Evaluation

The conceptual part of the modeling effort is concluded. The meta model is in place and validated. The next step is to implement the issue tracking part of the meta model, since, the effort of implementing a release history model on basis of CVS has already been made [FPG03].
The implementation of the issue tracking part of the meta model follows in essence the same principles and uses the same tools as the implementation of the CVS data model conducted earlier. By describing the issue tracking data model implementation, a possible implementation of the actual release history part of the meta model is being addressed as well. The upcoming section will describe the tools used for implementation. In the section "Implementation details" we will elaborate the implementation in more detail, by first introducing the overall idea of the process from loading and parsing XML files to storing objects in a database. Hereby, the issue tracking model will be linked with the CVS data model implementation. Finally the implementation section will discuss further improvements to the current implementation. The last section in this chapter will be concerned about the results of the implementation of the combined model. There, the evaluation will encompass results such as download and storage time of files, relevancy of the link between release history and issue tracking, etc. .

6.1 Technical overview

The requirements for this project state the use of Hibernate[1] and Eclipse[2]. Hibernate is an object/relational mapping tool for JAVA environments [Hib05]. It is an open source project and a component of the JBoss Enterprise Middleware System (JEMS[3]). Hibernate maps data representations of an object model to a relational data model with a SQL-based schema. It provides mapping of Java classes to database tables as well as query and retrieval facilities and relieves the developer from manual data handling in SQL and JDBC.
Hibernate is a very suitable tool for our implementation purpose. Namely, the implementation is concerned about translating an object oriented model into a database for data storage and retrieval. Hibernate does exactly that, it maps models onto relational database table schema. Further, a Java source code of an implementation does not have to be tempered with in order to use Hibernate. A developer using Hibernate should be only concerned with the model implementation in Java and data retrieval. The implementation is conducted using Eclipse.

[1] www.hibernate.org
[2] www.eclipse.org
[3] For more information, please see https://jboss.com

6.2 Implementation details

This section describes the complete implementation of the issue tracking data model. The description will encompass an overall view of the implemented classes and the complete process model of the implementation. To gain an insightful overview of the implementation idea, consider Figure 6.1.

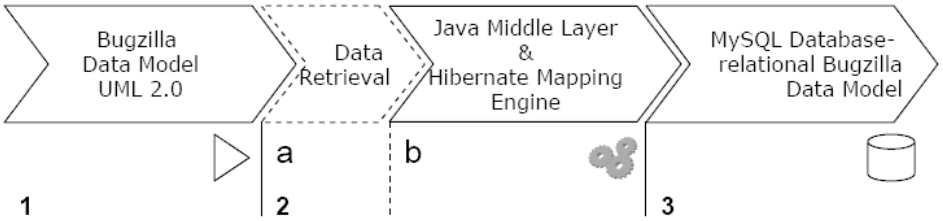


Figure 6.1: Implementation idea schematics

Under **1** there is the data model of an issue tracking system (Bugzilla) in form of a UML2.0 graphics. This object oriented model has to be implemented using Java and Hibernate. Hence, the following step under **2**is split into two parts. First, under **2a**, we retrieve actual data in bug reports on one of the bigger issue reporting sites[4].
After the data is retrieved, under **2b**, the UML classes are translated into Java Bean classes[5]. The Java Bean classes and the data retrieving classes (DOM Parser, Bug report loader), which well mention shortly, can be referred to as the Java middle layer of the implementation process. For each of the Bean classes, a Hibernate mapping file[6] is being made. The hibernate mapping file is used to construct the relational database tables and relations. Figure 6.6 shows a code snippet for the Bug class which represents implementation of the Issue entity of the issue tracking data model. at the bottom of the code, we see the get and set methods for but_ID, and the bug_number respectively. Figure 6.7 shows the corresponding XML mapping file for a Bug. With the "properties" tags, the attributes, defined by the get and set methods in the Bean class are being mapped. The "set" tags map the relations of the entities.
The Java bean classes and the corresponding Hibernate mappings designate the construct of the meta model as it will be present in a MySQL database (Figure 6.1 under **3**).
A bug report is usually stored as a XML file. This XML file holds all the relevant information in a predefined structure (Figure 6.8). Information is extracted by parsing the XML file. Since a bug report XML file has a static structure, where certain tags can be present or not, and since Java provides facilities to handle XML files, the documents were parsed using a DOM Parser (Document Object Model). A SAX parsing method could be used as well, but since we need all the tags and their information in a defined, static structure, the DOM parser was used. Further, the parsed bug report files are rather short, thus the construction of a DOM tree for a file is not significantly time consuming, if the processing time with DOM were taken as a disadvantage for that kind of document parsing. The DOM parser used here is extended with additional functionality. Namely, besides the parsing ability it also implements the Hibernate session facility for creating and stor-

[4]For the data source, the argoUML (www.argouml.tigris.org) issue repository was used. For testing purposes, however, the landfill.bugzilla.org repository was used.
[5]A Java bean class, as employed here, is a standard Java class, with get- and set- methods as shown in Figure 6.2 for each attribute of a entity in the UML model
[6]A Hibernate mapping file is an XML file that interprets Java Bean -class attributes in form of tags.

ing persistent objects in the MySQL database. The concept of Hibernate is described in [Hib05]. Concerning the retrieval of the XML files themselves, there are theoretically two options. The first implemented option is to directly reference a XML file over its URL and parse it directly, that is, on the Internet. The advantage of this method is, that no disc space is used for bug reports. However, this method would require a fast internet connection in order to function within affordable time. The second option, and the one, that is effectively used, is to first download and store the bug reports on a local hard drive and then access them for parsing. A significant advantage of this method is the minimal time consumption for accessing the files.
If the bug reports were not downloaded, the BugLoader.java file (Figure 6.2, a) is triggered first, to download and store Bug report files to a local hard-drive. The BugLoader retrieves XML files by their URL, then it reads the files and stores the XML code into local files with a designated name[7]. If the bug reports are present in a local folder on the hard-drive, the process starts by triggering the DOMBugParser.java file (Figure 6.2, b). The DOMBugParser first parses the selected bug reports for the bug number only, and stores the bug numbers in memory. Then it parses the bug reports again, whereby all data is extracted by parsing, the Hibernate session, provided by the HibernateUtil.java file (Figure 6.2, d) is created, Hibernate objects from the Java bean classes (Figure 6.2, c) are being initialized and stored after parsing. During the second parsing process, the dependencies for a bug are being compared to the bug numbers from the first parsing run. If we recall earlier elaboration, where it was stated, that bugs depend or block each other, the dependency information to a bug is the bug number of another bug that depends on or blocks the current bug. Therefore, we compare the dependency-bug numbers of a bug with all the numbers of bugs that are stored in the database to check, if a bug that appears in the dependency or blocking relation to a currently parsed bug is in the database or not. If a bug, that another bug depends on is not present in our MySQL database, it cannot be considered as a dependency or a blocking and it will be ignored by the parser until it is inserted in the database. In a 3rd parsing run, the DOMBugParser checks the links[8] between the issue tracking implementation and the CVS versioning data model. Hereby, the parser loads all Revision[9] objects from the database, provided, that the Versioning data is loaded first into the database. Then, the parser checks the commit messages in every revision for bug numbers by searching the text of the commit message. If a commit message points to a bug, that revision and the corresponding bug will be linked with a many-to-many relation[10]. Hibernate uses the mapping XML files (Figure 6.2, d) for each Java class to create the tables and relations in our MySQL database. For the establishment of the database connection and basic Hibernate configuration, the hibernate.cfg.xml file (Figure 6.2, d) is used. There, a user specifies the database name and appropriate credentials to access the database, the connection driver, the SQL dialect, the used hibernate mapping XML files and other settings[11]. Finally, hibernate creates the tables in the database and exits the current Hibernate session and the entire process.

The implementation is organized in three packages in the Eclipse project:

- *org.evolizer.base.issuetracking.hibernate*[12]: This package contains the DOM parser, the bug loader, the Hibernate mapping engine file and additional Hibernate configuration files.
- *org.evolizer.base.issuetracking.model*: The package contains all the Java Bean classes for each UML model entity.

[7] Bug report files are denoted as "bugreport**i**.xml, where by i is the respective number of the file.
[8] The linkage of the Issue and Revision entities was performed and described in [FPG03]
[9] As stated earlier, the Revision is the linking counterpart to the issue.
[10] A many-to-many relation is implemented as a set of objects (HashSet) that is added to each side of the relation. For more details, pleas see the Eclipse project
[11] please see the Eclipse project for mode details
[12] The org.evolizer.base project is the super set of the release history and issue tracking projects.

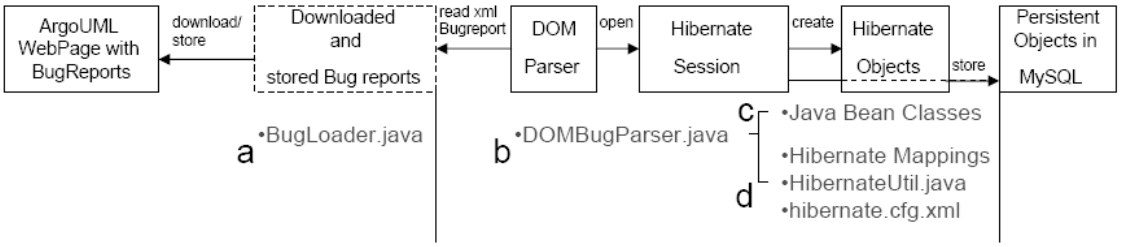


Figure 6.2: Detailed process graph for the issue tracking implementation

- *org.evolizer.base.issuetracking.model.mappings*: The Hibernate mapping files are stored in this package.

The downloaded bug reports are stored in a separate folder, but in the same Java project as the above mentioned packages. To access the bug reports, the DOM parser has to be set to the relative path of that folder to be able to access the bug report files.

6.3 Evaluation

The evaluation is concerned about the performance of the Bugzilla implementation as well as about problems that emerged during implementation. Another focus lies on the link between the Bugzilla implementation and the CVS data model in scope of the Evolizer project. For the purpose of evaluation, the ArgoUML issue repository was taken. For early testing purposes, the landfill.bugzilla.org repository was used.
The evaluation is concerned with indicating values regarding functionality and run-time characteristics of the implementation. Section 6.3.1 describes the intended results of the implementation. the intended results are the point of reference for the overall implementation status of the Bugzilla data model. The performance measurement procedure is implicitly denoted during the description of the achieved results, in section 6.3.2.

6.3.1 Requirements

The requirements concern the desired implementation level of the issue tracking data model. Further, requirements relate to the desired functionality in terms of retrieval of relevant data, like relations between entities, such as the relation of Issue and Comment or Issue and Activity.

The implementation level is defined by the following requirements:

- Implementation is conducted using Java and Hibernate;
- The Bugzilla data model as described in Figure 3.6 is implemented in all its entities and relations, whereby adjusted to the ArgoUML issue repository. ;
- The Bugzilla implementation is capable of retrieving issue reports from any problem report site on the web. Hereby an entire web repository of a problem report system can be locally stored and the data extracted;
- The process of the Bugzilla implementation, as described in section 6.2 is fully functional;

- The Bugzilla implementation is easily explicable and appropriate for successive developers to use;
- The Bugzilla implementation performs in acceptable time frames;
- The Bugzilla implementation is capable of linking its data to the appropriate entities in the CVS implementation in scope of the Evolizer project according to the description in section 6.2;

The intended functionality is designated in form of representative feasible queries that demonstrate the usability and effectiveness of the implementation and it is described as follows:

Basic queries:

- Retrieve Issue number
- Retrieve Issue Activities
- Retrieve Reporter
- Retrieve Assignee
- Retrieve Comments
- Retrieve Person as Comment Creator
- Retrieve Person as Activity Creator
- Retrieve Attachments
- Retrieve all Persons to an Issue
- Retrieve Issue Status/Severity/Milestones/Keywords
- Retrieve Issue triggered by Revision

The performance is measured in the time it takes to store a certain number of issues. The intended minimal time it takes the implementation to process a problem report set depends on the number of reports. The link between issue tracking and versioning is measured in the number of effectively established links between Issue and Revision.

6.3.2 Achieved and measured results

Implementation Level Evaluation

The implementation level was achieved according to the requirements in the following measure:

- *Implementation is conducted using Java and Hibernate*: Java as a programming language and Hibernate were used extensively to implement the Bugzilla data model.
- *The Bugzilla data model as described in Figure 3.6 is implemented in all its entities and relations, whereby adjusted to the ArgoUML issue repository*: The Bugzilla problem-report-XML files (Figure 6.5) have different tags in comparison to the ArgoUML XML-files, which were used for the evaluation. In order to process the ArgoUML files, the DOMBugParser.java had to be set to parse XML-files according to ArgoUML notation instead of the landfill.bugzilla.org-notation. The modifications concerned changing XML tag-names in the parser. All entities from the Bugzilla data model are applicable to the ArgoUML data representation.

- *The Bugzilla implementation is capable of retrieving problem reports from any problem report site on the web. Hereby an entire web repository of a problem report system can be locally stored and the data extracted*: The BugLoader.java can be set to the URL of any web-based problem tracking repository, as long as this repository is conform to the common problem report notation as in Bugzilla or ArgoUML (Figure 6.5).
- *The process of the Bugzilla implementation, as described in section 6.2 is fully functional*: All the designated components are implemented and related to each other as described.
- *The Bugzilla implementation is easily explicable and appropriate for successive developers to use*: The implementation combines the parsing of XML-files and the creation and storage of Hibernate objects in one file (DOMBugParser.java). This method of implementation has its advantages, since the only run-time component is the DOMBugParser.java and the entire process is capsuled in one class. A more proper approach is to externalize the entire Hibernate mapping engine for better understandability and debugging of the implementation.
- *The Bugzilla implementation performs in acceptable time frames*: The processing of the entire ArgoUML repository (3'868 Issues as of the 17.01.2005) takes approximately 28 minutes. The detailed time allocation is shown in the "Performance Evaluation" subsection, further below.
- *The Bugzilla implementation is capable of linking its data to the appropriate entities in the CVS implementation in scope of the Evolizer project according to the description in section 6.2* The linking of the Revision and Issue entities with an association is feasible and performs as a method of the DOMBugParser.java. The resulting set is an association table in the data base showing Issue IDs and the corresponding Revision IDs.

Functionality Evaluation

The designated queries (see Basic Queries above) can be fully performed to retrieve the relevant data for analysis. The Basic Queries are executed externally, in native SQL. A further possibility is to internalize some basic queries into the Hibernate querying facility.
By querying the database with the CVS and ArgoUML data for the linkage between Revision and Issue 300 links from Issue to the keyword "Issue Number: " in Revision were made out of 6'138 Revisions and 3'868 Issues. From these 300 possible links, 65 links were effective matches of Revision and Issue. The 300 possible links are represent the matches of the keyword "Issue Number:" only, without an effective issue number following it. The 65 effective links represent the matching of the keyword and the corresponding issue number. A related keyword search and linkage of Revisions and Issues is conducted in [FPG03].

Performance Evaluation

The DOMBugParser.java is the only component that performs in run-time. The time (in milliseconds) for the entire process is allocated to the methods in the DOMBugParser.java. For performance testing, the Profiler[13] plug-in for Eclipse was used. The result set of the performance evaluation shows information as in Figure 6.3.

From Figure 6.3 we can see, that the time allocation is unevenly spread across the threads. The most processing time is consumed by the preLoadDependencies and loadParseComments threads. The preloadDependencies thread goes in the first parsing run. The second and main parsing run, where all the data is parsed and stored allocated the overall time mostly to the

[13] http://eclipsecolorer.sourceforge.net/index_profiler.html

Name	Calls	%	Time	%	Time/Call	Total time	Inst. time
DOMBugParser.preLoadDependencies	1	0.24	5062	86.19	5062.000000	5062	500
DOMBugParser.main	1	0.24	0	0.00	0.000000	0	500
DOMBugParser.loadParseVersion	18	4.31	0	0.00	0.000000	0	500
DOMBugParser.loadParseTimeStamp	18	4.31	16	0.27	0.888889	16	500
DOMBugParser.loadParseStatWhiteboard	18	4.31	15	0.26	0.833333	15	500
DOMBugParser.loadParseRsolutionStatus	18	4.31	0	0.00	0.000000	0	500
DOMBugParser.loadParseReporter	18	4.31	0	0.00	0.000000	0	500
DOMBugParser.loadParseProduct	18	4.31	141	2.40	7.833333	141	500
DOMBugParser.loadParsePriority	18	4.31	0	0.00	0.000000	0	500
DOMBugParser.loadParsePlattform	18	4.31	0	0.00	0.000000	0	500
DOMBugParser.loadParseOS	18	4.31	15	0.26	0.833333	15	500
DOMBugParser.loadParseMilestone	18	4.31	31	0.53	1.722222	31	500
DOMBugParser.loadParseKeyword	18	4.31	0	0.00	0.000000	0	500
DOMBugParser.loadParseDependency	18	4.31	0	0.00	0.000000	0	500
DOMBugParser.loadParseDeltaTs	18	4.31	0	0.00	0.000000	0	500
DOMBugParser.loadParseComponent	18	4.31	154	2.62	8.555556	154	500
DOMBugParser.loadParseComments	18	4.31	361	6.15	20.055556	361	500
DOMBugParser.loadParseClassification	18	4.31	0	0.00	0.000000	0	500
DOMBugParser.loadParseCc	18	4.31	16	0.27	0.888889	16	500
DOMBugParser.loadParseBugSeverity	18	4.31	0	0.00	0.000000	0	500
DOMBugParser.loadParseBugFileURL	18	4.31	0	0.00	0.000000	0	500
DOMBugParser.loadParseBugDescription	18	4.31	0	0.00	0.000000	0	500
DOMBugParser.loadParseBlocking	18	4.31	0	0.00	0.000000	0	500
DOMBugParser.loadParseAttachment	18	4.31	31	0.53	1.722222	31	500
DOMBugParser.loadParseAssignees	18	4.31	31	0.53	1.722222	31	500
DOMBugParser.linkRevisionAndBug	0	0.00	0	0.00	0.000000	0	500
DOMBugParser.<init>	1	0.24	0	0.00	0.000000	0	500
DOMBugParser.<clinit>	1	0.24	0	0.00	0.000000	0	500

Figure 6.3: Thread table for the DOMBugParser.java; *source: Eclipse Profiler plug-in*

loadParseComments thread, since issues are known to have more than one comment or attachment(see the time allocation for loadParseAttachments in Figure 6.3).

Figure 6.4 further illustrates the time allocation and shows the thread call procedure for the most frequently called threads in the DOMBugParser.java.
The memory usage during the process is shown in Figure 6.5. The peak in performance use is caused by the initial call of the DOMBugParser.java class. From then, the memory usage declines. Due to problems within the Profiler plug-in for Eclipse, which was used to measure the performance of our Bugzilla implementation, Figures 6.3, 6.4 and 6.5 show the performance of the entire implementation for only 20 thread calls (20 bug reports). The time allocation percentage is approximately constant for every number of bug reports for the following threads, since the number of instances generated by those threads per bug-report is constant[14]:

- loadParseTimestamp thread: 0.27% of 28 min makes 4.536 seconds
- loadParseStatWhiteboard thread: 0.26% of 28 min makes 4.368 seconds
- loadParseProduct thread: 2.40% of 28 min makes 40.32 seconds
- loadParseOS thread: 0.26% of 28 min makes 4.368 seconds
- loadParseComponent thread: 2.62% of 20 min makes 44.016 seconds

Despite, that the above mentioned threads generate each the same number of instances per bug, their time consumption is different. This variation in time consumption is due to the fluctuation in the internet connection of the machine, our implementation runs. The parsing of the XML-files invokes the parsing of the DTD (Document Type Definition) for every bug report.
The overall performance of the implementation is satisfying, although improvements can be

[14]For instance, a bug report has one and only one time-stamp to designate its creation time

made. Especially the download of the XML files takes long time and should be shortened. Further, as seen in Figure 6.4, some threads (the dark marked thread on the top left of the Figure) still consume lot more time than other. This is not a drastic performance lack but, it might be considered when processing larger web problem reporting projects.

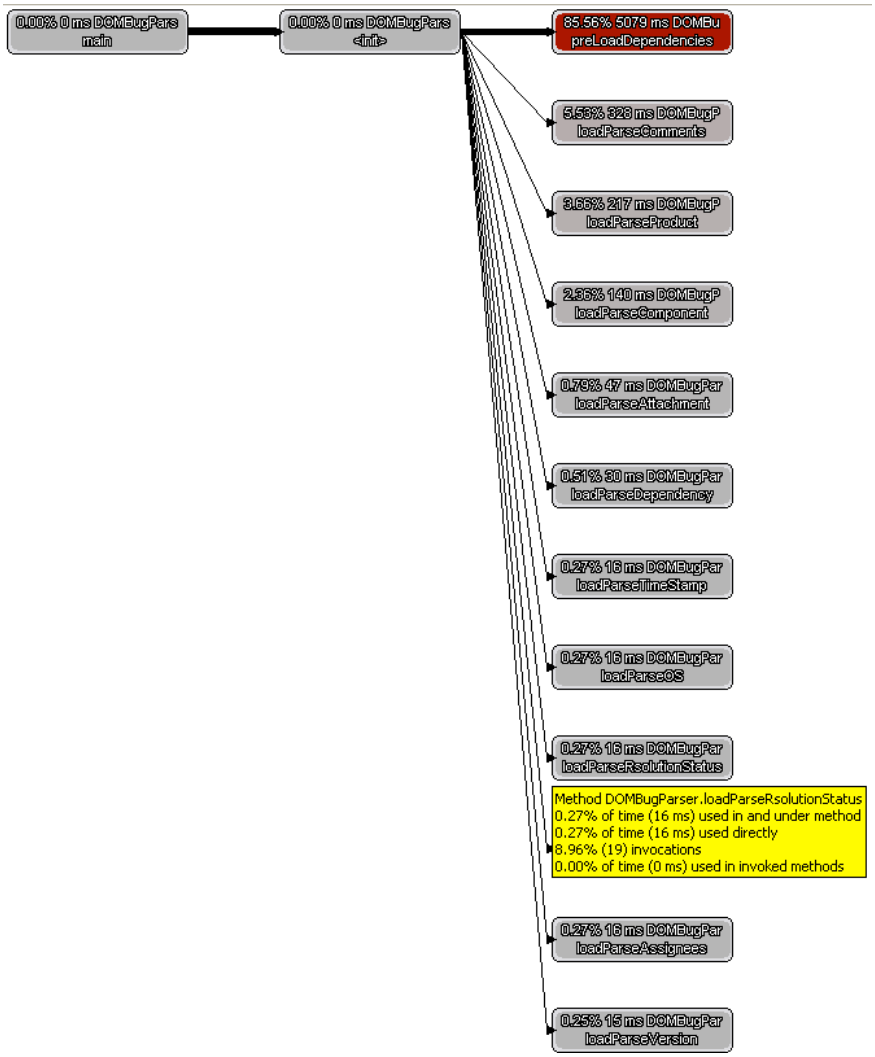


Figure 6.4: Thread call graph for the DOMBugParser.java class; *source: Eclipse Profiler plug-in*

6.3.3 Problems during implementation and evaluation

Implementation Problems

As mentioned earlier, the designation of a problem report is different in ArgoUML and in Bugzilla; ArgoUML refers to problems as issues, whereby Bugzilla refers to them as bugs. This means, that the XML files that hold the information about an issue or bug are slightly different in their notation. This further implies, that the parser had to be reset to ArgoUML notation in the evaluation

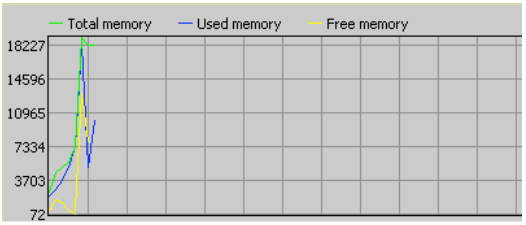


Figure 6.5: Memory usage of the DOMBugParser.java; *source: Eclipse Profiler plug-in*

```
import java.util.Date;

public class Bug {

    private Long id;
    private Long bug_number = PreLoader.bugIDLoader;
    private String url = PreLoader.bug_file_locLoader;
    private String summary = PreLoader.bugSummaryLoader;
    private String status_whiteboard = PreLoader.status_whiteboardLoader;
    private String keyword = PreLoader.keywordsLoader;
    private String priority = PreLoader.priorityLoader;
    private String severity = PreLoader.bug_severityLoader;
    private String description;
    private String date_opened = PreLoader.creation_tsLoader;
    private Component component;

    private Set <Milestone> milestones = new HashSet<Milestone>();
    private Set <Keywords> keywords = new HashSet<Keywords>();
    private Set <ComputerSystem> compSystems = new HashSet <ComputerSystem>();
    private Set <Bug_state> bugStates = new HashSet<Bug_state>();
    private Set <Dependency> dependencies = new HashSet<Dependency>();
    private Set <Attachment> attachments = new HashSet<Attachment>();
    private Set <Comment> comments = new HashSet<Comment>();
    private Set <Person> persons = new HashSet<Person>();
    private Set <BugActivity> bugActivities = new HashSet<BugActivity>();

    Bug() {}

    public Long getId() {
        return id;
    }

    private void setId (Long id) {
        this.id = id;
    }

    public Long getBug_number() {
        return bug_number;
    }
```

Figure 6.6: Code snippet of a Java bean class;

phase.

Another problem that raised during implementation is the parsing methodology itself for the XML-files. Two aspects of the parsing problematics were identified. First off, the XML files rely on defined DTDs which caused problems during the download and later during the processing of the XML files. The files themselves could sometimes not be read, because the DTD was incorrect or the BugLoader.java downloaded and stored an empty file (in 3'868 files there were 220 empty files, which makes 5,68%). This caused the parser to throw error messages and stop the

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
<class name="Bug" table="BUGS">
    <id name="id" column="BUG_ID">
        <generator class="increment"/>
    </id>
    <property name="date_opened" column="BUG_OPENED"/>
    <property name="bug_number" type="long"/>
    <property name="url"/>
    <property name="summary"/>
    <property name="status_whiteboard"/>
    <property name="keyword"/>
    <property name="priority"/>
    <property name="severity"/>
    <property name="description"/>

    <set name="milestones" table="BUG_A_MILESTONE" inverse="true">
        <key column="BUG_ID"/>
        <many-to-many column="MILEST_ID" class="Milestone"/>
    </set>
    <set name="keywords" table="BUG_A_KEYWORDS" inverse="true">
        <key column="keyword"/>
        <many-to-many column="KEYWORD_ID" class="Keywords"/>
    </set>
    <set name="compSystems" table="BUG_A_COMPSYS" inverse="true">
        <key column="BUG_ID"/>
        <many-to-many column="COMPSYS_ID" class="ComputerSystem"/>
    </set>
    <set name="bugStates" table="BUG_A_BUGSTATE" inverse="true">
        <key column="BUG_ID"/>
        <many-to-many column="BUG_STATE_ID" class="Bug_state"/>
    </set>
```

Figure 6.7: Code snippet of a Hibernate mapping file

parsing. This problem was solved by reformating each stored bug report, so the XML code could be correct and well-formed and parsed properly. The other aspect of the mentioned problematics was immanent to earlier versions of the parser. Since the DOM parsing method was used, every tag in the XML document had to be designated as a node in the DOM tree. As long as the tag to be parsed was present in the document, the parser functioned well. As soon as the tag was not present -for instance, if the currently parsed bug did not depend on any other bug, the dependency tag was missing- the parser stopped and threw error messages. This problem was solved by improving the parser, enabling it to recognize missing tags and ignore them.

In the implementation level evaluation, we stated, that the parsing and the Hibernate mapping storage of objects is done by the same class. In early versions of the implementation, the mentioned processes were separated. The parser did only the parsing, whereby another class incorporated the entire Hibernate engine facilities. The problem with this first approach, was, that relations among certain entities of the data model did not establish correctly. E.g., Bug 1 has 50 comments attached to it. Bug 2 has 3 comments attached to it. Normally, the Comments-table in the database should show the 50 first comments with the number of the Bug 1 as foreign key, the 51st, 52nd and 53rd comment should then have the number of Bug 2 as foreign key. The mentioned approach did not manage to designate the comments correctly. It showed all 53 comments as comments of the Bug 1. Thus, the implementation was modified, so that the creation of Hibernate objects and the relation-mapping occurred during the parsing of a bug repor. The problem of incorrect relations was solved hereby, but the implementation lost on flexibility.

```
<?xml version="1.0" standalone="yes" ?>

<bugzilla version="2.21" urlbase="http://landfill.bugzilla.org/bugzilla-tip/"
maintainer="webmaster@bugzilla.org">
 <bug>
  <bug_id>1</bug_id>
  <creation_ts>2000-06-15 11:35</creation_ts>
  <short_desc>THIS IS A VERY DECENT DESCRIPTION, THANK YOU!</short_desc>
  <delta_ts>2005-09-03 02:32:05</delta_ts>
  <reporter_accessible>1</reporter_accessible>
  <cclist_accessible>1</cclist_accessible>
  <classification_id>1</classification_id>
  <classification>Unclassified</classification>
  <product>MyOwnBadSelf</product>
  <component>comp2</component>
  <version>unspecified</version>
  <rep_platform>PC</rep_platform>
  <op_sys>Windows 2000</op_sys>
  <bug_status>VERIFIED</bug_status>
  <resolution>WONTFIX</resolution>
  <bug_file_loc>javascript:alert("THERE WILL BE NO PICTURES OF PIGS SHOOTING DOWN BROTH
  <status_whiteboard>Is this freeform and how big is it? IF it goes on forever then tha
  <keywords>KeyMe</keywords>
  <priority>P5</priority>
  <bug_severity>enhancement</bug_severity>
  <target_milestone>---</target_milestone>
  <dependson>873</dependson>
  <dependson>1087</dependson>
  <dependson>2029</dependson>
  <dependson>2030</dependson>
  <dependson>2290</dependson>
```

Figure 6.8: Code snippet of a Bugreport xml file

The link between Revision and Issue entity is, as described earlier, implemented as a many-to-many relation. The problematics of this relation was, that there is no explicit and defined interface or linking of both CVS and Issue-tracking data models. The linking attributes are the commit message in the Revision, and the issue number in the Issue. The commit message is a free text with no structure or pre-defined keywords, where as the bug number has no further information to it. The commit message had to be searched for and indication to a bug number [FPG03]. The problem was partially solved, by instantiating the Revision's commit message as an Object in the DOMBugParser letting the parser search the commit message for every issue and then, if a match was found, assign the current Revision to the Issue. The searching method in the commit message is capable of identifying only a pre-defined character sequence, which makes it more in-appropriate for wider use.

Evaluation problems

During evaluation, the main problem was the mentioned Profiler plug-in for Eclipse. The calculation of the time allocations to particular threads that were invoked during run-time was conducted on the maximum number of parsed files that could be handled by the Profiler plug-in. The overall process time was however not tangented by that problem- our implementation was put to work in stand-alone mode to retrieve the process time.
In essence, no more significant problems emerged during the evaluation, except that the ArgoUML site was very slow and hardly accessible.

Chapter 7

Conclusions

Release history management has increasingly been the subject of research and development in recent years. The notion of history in software systems has become more immanent to software development and reserves a constant place in this domain. In this thesis, the notion of history, based on a meta model approach to various revision history data models was conceptualized and implemented using sophisticated tools. The release history meta model is an approach where different models for the history concept were taken as a base for another data model -a meta model- that commonly describes all the taken models and incorporates all the needed semantics to describe other versioning data models as well. To provide a meta model for release history as in this thesis brings several problematics in different areas of modeling and implementation that need to be addressed and solved. The conceptual issues start by first designating a modeling technique and the degree and way of information abstraction. Further, the extraction of relevant information and the designation of model entities posed an interesting task, as well as the referencing to the base-data models. A special concern was thus given to the structured comparison of the data models and the identification of similarities among the reference- data-models. In order to successfully map an effective meta model, the derivation of information from the reference data models had to be accurate and consistent. For the meta model construction to be accurate and effective, all reference data models were modeled in the same way -the data description and abstraction was the same for all models- and the meta model was derived entity by entity from those data models by identifying same or similar core versioning concepts and translating them to entities and relations of the meta model.
A further problem in modeling in general is the combination of different models into one. The release history meta model was extended in its information wealth by the issue tracking data model derived from a well known and widely used problem tracking system. The issue tracking aspect is a closely related domain to release history and by adding that aspect to the meta model we have extended the descriptive capability on to issue or change request tracking. The combination of these two models posed a different problem aspect. Namely, the issue tracking data model has dynamics of its own, so, it was an interesting task to add the issue-tracking model with minimal modifications and a proper and functional link to the release history meta model. Conceptually, this extension was about adding a bidirectional association between entities of each model and the implementation was realized as the combination of two independent projects.
The tools used for modeling and implementation provided the needed functionality to accomplish the implementation work. The implementation is at a stage, where the release history model is functionally combined with the issue tracking project and where data can be analyzed in a MySQL database. The actual release history implementation was done on CVS in the scope of a related project. The CVS data model is a reference to the meta model. The meta model is further capable of modeling CVS as well, meaning that the implementation of the release history concept

in CVS corresponds to a possible implementation of the meta model.

The modeling effort conducted in this thesis is to give a base for further development in meta modeling release history. Meta modeling approaches to release history are not many. An explicit meta model for release history is not in place yet and the meta model developed in this thesis could be considered a sort of a point of reference for future work. If an overall meta model can be put in place at all is to be discussed, since the notion of history changes and second, new modeling concepts in software systems in general emerge. So it would be difficult to establish a generally valid meta model without having the problem of information handling and conceptual-difference understanding. However, the meta modeling effort conducted in this thesis can be as well understood as a concept providing a solid base for modeling efforts in related or even non related areas to software development.

References

[BCS05] C. Michael Pilato Ben Collins-Sussman, Brian W. Fitzpatrick. Version control with subversion 1.1. 2005.

[Cap03] A. Capiluppi. Models for the evolution of os systems. In *Proceedings of the International Conference on Software Maintenance(ICSM 2003)*, pages 65–74, 2003.

[Cap04] A. Capiluppi. Evolution of understandability in oss projects. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 58–66, 2004.

[CC03] J. Nagra J. Pitts K. Wampler C. Collberg, S. kobourov. A system for graph-based visualization of the evolution of software. In *Proceedings of the 2003 ACM Symposium of Software Visualization*, pages 77–86, 2003.

[Ced05] Per Cederquist. Version management with cvs. 2005.

[Fis05] J. Fischer. Metamodelierung. 2005.

[Fow04] Martin Fowler. *UML Distilled, A brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 2004.

[FPG03] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a Release History Database from Version Control and Bug Tracking Systems. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM)*, pages 23–32, Amsterdam, The Netherlands, September 2003. IEEE, IEEE Computer Society.

[GT05] K. Ehrig G. Taenzler. Visuelle sprachen projekt, einfhrung. 2005.

[HH04] Ahmed E. Hassan and Richard C. Holt. Predicting change propagation in software systems. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 284–293, Washington, DC, USA, 2004. IEEE Computer Society.

[Hib05] Hibernate- relational persistence for idiomatic java. 2005.

[Jaz02] M. Jazayeri. On architectural stability and evolution. In *Reliable Software Technologies-Ada-Europe 2002*, pages 13–23, 2002.

[JB05] Jr. S. Kim M. Godfrey J. Bevan, E. J. Whitehead. Faciliating software evolution research with kenyon. 2005.

[LB85] M.M. Lehman and L. Beladi. *Program Evolution - Processes of Software Change*. London Academic Press, 1985.

[ML02] S. Ducasse M. Lanza. Understanding software evolution using a combination of software visualization and software metrics. In *Proceedings of LMO 202 (Langages et Modles Objets)*, pages 135–149, 2002.

[PH] Frantishek Plasil Petr Hnetynka. Distributed versioning model for mof.

[Rat01] Rational. Rational clearcase family documentation supplement. 2001.

[Rat03a] Rational. Clearcase and clearcase lt introduction (unix/windows edition). 2003.

[Rat03b] Rational. Rational clearcase and clearcase lt command reference (unix/windows edition). 2003.

[SD01] S. Ducasse S. Demeyer, S. Tichelaar. Famix 2.1 - the famoos information exchange model. 2001.

[TB96] S. G. Eick T. Ball. Software visualization in the large. 29(4):33–43, 1996.

[Tea05] The Bugzilla Team. The bugzilla guide - 2.19.3 development release. 2005.

[TG05] Stephane Ducasse Tudor Gibra, Jean-Marie Favre. Using meta-model transformation to model software evolution. 2005.

[TZ04] S. Diehl A. Zeller T. Zimmermann, P. Weissberger. Mining version histories to guide software changes. In *26th International Conference on Software Engineering (ICSE 2004)*, pages 563–572, 2004.

[XW04] M.-A. Storey X. Wu, A. Murray. A reverse engineering approach to support software maintenance: Version control knowledge extraction. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 90–99. IEEE, IEEE Computer Society, 2004.