An Abstraction for Version Control Systems

Matthias Kleine, Robert Hirschfeld, Gilad Bracha

Technische Berichte Nr. 54

des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam | 54

Matthias Kleine | Robert Hirschfeld | Gilad Bracha

An Abstraction for Version Control Systems

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.de/ abrufbar.

Universitätsverlag Potsdam 2012
http://info.ub.uni-potsdam.de/verlag.htm

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Das Manuskript ist urheberrechtlich geschützt.

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
URL http://pub.ub.uni-potsdam.de/volltexte/2012/5562/
URN urn:nbn:de:kobv:517-opus-55629
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-55629

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-158-5

Version Control Systems (VCSs) allow developers to manage changes to software artifacts. Developers interact with VCSs through a variety of client programs, such as graphical front-ends or command line tools. It is desirable to use the same version control client program against different VCSs. Unfortunately, no established abstraction over VCS concepts exists. Instead, VCS client programs implement ad-hoc solutions to support interaction with multiple VCSs.

This report presents Pur, an abstraction over version control concepts that allows building rich client programs that can interact with multiple VCS. We provide an implementation of this abstraction and validate it by implementing a client application.

Contents

1 Introduction **9**
- 1.1 Contributions 10
- 1.2 Report Structure 10

2 Background **11**
- 2.1 Choice of Version Control Systems 11
- 2.2 Version Control System Architectures Compared 12
- 2.3 Requirements 28
- 2.4 Summary 30

3 Pur—An Abstraction for Version Control **31**
- 3.1 Stores and Snapshots 32
- 3.2 Versions 32
- 3.3 Historians and Repositories 32
- 3.4 Pur by Example 32
- 3.5 Summary 35

4 Implementing Pur for Concrete Back-ends **37**
- 4.1 Abstract Implementation 37
- 4.2 Implementation for Back-Ends 42

5 Pur for Newspeak—PNS **47**
- 5.1 Snapshots 47
- 5.2 Stores 48
- 5.3 Diffing Algorithm 50
- 5.4 User Interface 51
- 5.5 Outlook 56
- 5.6 Summary 58

6 Evaluation **59**
- 6.1 Provide Rich Semantics 59
- 6.2 Version Control System-agnostic Interface 59
- 6.3 Minimal Interface 60

6.4 State-based Non-linear History Model 60
6.5 Consistent Branching Model 61
6.6 Conclusion 61

7 Related Work **63**
7.1 Software Configuration Management 63
7.2 Version Control 64
7.3 Implementations of Version Control System Abstractions . . 66

8 Summary and Outlook **73**

Bibliography **75**

List of Figures

2.1 A simplified view of Subversion's architecture 14
2.2 Git and Mercurial unify repositories and clients 15
2.3 Representing history as a directed acyclic graph of snapshots . 16
2.4 Cherry picking the changes made by b onto x 17
2.5 Example Subversion revisions visualized 18
2.6 A simplified view of Git's object model 21
2.7 A simplified view of Mercurial's history model 22
2.8 A simplified model of the revlog abstraction used in Mercurial 22
2.9 Exemplary comparison of Mercurial's and Git's history models 23
2.10 An example of upstream branches in Git 27

3.1 Interfaces for the objects being versioned by Pur 31
3.2 Visualization of snapshots 33
3.3 Visualization of a history graph. Two versions shown in detail . 34
3.4 Historians create new versions 35

4.1 Abstract classes provided by framework 39
4.2 Specializations for local repositories 40
4.3 Specializations for remote repositories 40

5.1 Implementation of snapshot interface for Newspeak 48
5.2 Implementations of store interface for Newspeak 48
5.3 Existing and required synchronization mechanisms 49
5.4 Classes of the diffing algorithm 51
5.5 Screenshot of PNS 52
5.6 Interaction choices between image and current historian 53
5.7 Changes were expanded and a commit message was entered . 53
5.8 Interaction with the current historian and related historians . . 54
5.9 Interaction with other local historians 55
5.10 Interaction with remote repositories 55
5.11 Excerpt of the merge UI 56

7.1 Eclipse's hierarchical diffing UI 67
7.2 IntelliJ IDEA's VCS-interaction menus for Subversion and Git . 68

List of Tables

2.1 Example for status of working copies 19
2.2 Subversion's mergeinfo illustrated 25

4.1 Comparison of corresponding VCS commands 43
4.2 Implementation of extension commands 44

1 Introduction

Version Control Systems (VCSs) help developers to manage changes to software artifacts. They allow developers to document, share, and merge changes. These tasks are critical to the success of software projects, because they are both frequently recurring and prone to human error. Software is built by teams of developers who work in parallel on the same source code. The need to share and merge changes is thus omnipresent. Without the assistance of VCSs, developers can fail to communicate their changes and must merge changes manually; changes become untraceable. VCSs address these problems by allowing developers to share their changes in a structured and traceable way. Changes are recorded in the VCS and conflicts can be solved by considering the history of changes. The potential for error is thus reduced.

Developers interact with VCSs through a variety of client programs. Such client programs include command line interfaces, graphical front ends, Integrated Development Environments (IDEs), project management web applications, or information extraction tools, such as refactoring reconstruction systems. These client programs enhance the VCSs by use-case specific functionality, for example by providing a certain kind of user interface, or by extracting certain data.

It is desirable to use the same version control client program against different VCSs. The benefits provided by a client program are often applicable across different VCSs. For example, user interfaces that are provided by IDEs, such as those to browse a project's history or to show differences between versions are, to a large extent, independent of the concrete VCS being used. This applies to other client programs, such as project management web applications or graphical front-ends.

Unfortunately, no established VCS abstraction exists. Client programs that want to interact with multiple VCSs face the difficulty that VCS interfaces differ in terminology and concepts. These differences are addressed by ad-hoc solutions that cannot be re-used across client programs. This report contributes to the solution of this problem with the following:

1.1 Contributions

An Abstraction for Version Control We describe Pur, an abstraction for VCSs that captures concepts common to the VCSs Subversion, Git, and Mercurial. This abstraction is intended to maintain a sufficiently rich set of concepts so as to serve as a basis for generic VCS client programs.

Version Control Abstraction in Newspeak We validate the practicability of the abstraction by implementing it for Git and Mercurial in the Newspeak programming language. Furthermore, we describe how to implement it for Subversion.

Version Control Application in Newspeak We validate the applicability of the abstraction by implementing a version control application that provides a user interface to interact with Pur.

1.2 Report Structure

The report is structured as follows: First, chapter 2 analyzes and compares the three VCSs Subversion, Git, and Mercurial. Based on this analysis it identifies requirements that must be satisfied by a common abstraction. The resulting common abstraction Pur is presented in chapter 3. Next, chapter 4 shows how this abstraction can be implemented in the Newspeak programming language. Chapter 5 describes the implementation of the version control application PNS that makes use of the Pur implementation. Based on these implementations, chapter 6 evaluates Pur in respect to the requirements initially identified. Next, chapter 7 compares Pur to other approaches to abstract over version control. Chapter 8 sums up the findings of this report and gives an outlook on future work.

2 Background

This chapter provides the background for the version control abstraction Pur. It presents the VCSs that Pur must support, identifies common concepts of these systems, and extracts requirements that must be met by a common abstraction.

2.1 Choice of Version Control Systems

The applicability of a version control abstraction depends on the set of VCSs that it supports. The set of supported systems itself depends on the choice of VCSs that are analyzed to establish requirements for Pur. The selection of such VCSs can be made with several criteria in mind. An analysis of a large number of VCSs can guarantee that a wide selection of concepts is considered but will at the same time reduce the resources available for the analysis of each single system. In contrast, the opposite approach of analyzing a small number of VCSs can guarantee a higher depth of analysis for each system but can only produce relevant results if the selection of VCSs is relevant to many users.

The selection of VCSs that is analyzed for Pur is restricted to a small set of popular VCSs. We identify popular VCSs by surveying recent research and actual usage of VCSs. In 2007, Apache Subversion was found to be the leader in the standalone Software Configuration Management (SCM) market [Sch07], where SCM provides the wider context for version control [MWE10]. While no research known to us suggests that Subversion's prevalence has declined in proprietary software development, a strong trend towards Distributed Version Control Systems (DVCSs) can be observed in open source projects [DAS09]. Unlike centralized systems, such as Subversion, DVCSs replicate history across repositories [OG90, Car98]. Big projects such as the Linux kernel, Google's Android, Qt, or VLC now use Git[1] and projects such as Python, OpenOffice, or Vim now use Mercurial[2].

[1] `https://git.wiki.kernel.org/index.php/GitProjects` – last checked 01.12.2010
[2] `http://mercurial.selenic.com/wiki/ProjectsUsingMercurial` – last checked 01.12.2010

In open source projects, Subversion, Git and Mercurial appear to be the most widely used systems. This is indicated by the number of projects found on open source hosting platforms. The website Ohloh[3] collects statistics on open source projects. It suggests that over 70 % of projects are using Subversion or CVS, over 25 % use Git, followed by more than 2 % using Mercurial. These statistics suggest that by choosing Subversion, Git, and Mercurial the version control concepts needed by most projects can be regarded in the analysis. This is also suggested by the results reported by [Mal10]. Consequently, we analyze Git, Subversion, and Mercurial to form the basis of Pur.

2.2 Version Control System Architectures Compared

This section compares the three systems Git, Mercurial, and Subversion in order to establish requirements for a common abstraction. The comparison is structured by aspects that are relevant across the three systems. This section gives a brief introduction of these aspects, followed by a comparison of the VCSs along these aspects.

VCSs allow developers to manage changes to software artifacts. Software artifacts whose changes are tracked are said to be **versioned**. The chosen VCSs expose software artifacts to the user as a file hierarchy. The history of this file hierarchy is stored in **repositories**. In the chosen VCSs, repository architectures fall into two categories. Subversion stores the history of a project in a single central repository. Git and Mercurial allow replicating the history across multiple repositories.

All three VCSs allow developers to access versioned files through **working copies**, also known as work spaces [Est96]. A working copy is a copy of the versioned directory hierarchy that is under control of a developer. Working copies additionally store meta data that indicates what repository the working copy corresponds to and how the files in the working copy relate to the history stored in that repository.

Working copies allow recording changes. A developer can modify files in a working copy without affecting the repository. Changes are only transferred to the repository and recorded in its change history on explicit request by the developer. This action is often called “commit” and usually requires the developer to enter a message describing the changes. Uncommitted local changes can be undone by restoring the working copy to a previous state that is stored in the repository.

[3] http://www.ohloh.net – last checked 11.08.2011

When changes are made in more than one working copy in parallel, development is said to **branch**. Depending on the changes being performed, it may be desirable to keep branches separate for a certain time, and thus version the history of branches separately. Typical scenarios for this are branches that contain new and unstable features and are therefore not to be used by anyone but those developing these features.

A **history model** specifies how the history of changes to software artifacts is represented in a VCS. The history models employed by the three VCSs fall into two distinct categories. Git's and Mercurial's history models have inherent support for representing branching development. In contrast, Subversion represents branches on top of the history model.

On top of the history model, the **branching model** specifies how branches are exposed as named entities. The branching model can be distinguished from the history model. A history model may be able to represent the existence of branches, but does not specify how branches are exposed as named entities. Being able to identify branches by name is often desirable, as it allows developers to communicate interaction between branches. For example, it might be desirable to ask the VCS to merge the changes from the "development" branch into the "stable" branch. A clear distinction of history- and branching model can only be found in Git and Mercurial. It is nevertheless desirable to distinguish branching- and history model, as it simplifies comparing Git and Mercurial, which have near-identical history models but diverging branching models. The following sections analyze how the three VCSs differ in the way that they address the various aspects.

2.2.1 Repositories

The three VCSs exhibit two distinct ways of organizing repositories. In Subversion, a project has exactly one repository whereas in Git and Mercurial, each developer has his own repository. Centralized and decentralized repositories have different properties. Centralized repositories allow better control over how and by whom history is interacted with. In contrast, decentralized repositories replicate history and thus allow developers to version their personal changes without having access to a shared repository. They thus facilitate distributed development. Replicated history additionally allows many operations to perform faster but requires additional tools to synchronize history.

Subversion Subversion's architecture is based on a client-server model. As shown in fig. 2.1, a project that is versioned using Subversion has a central repository, which can be accessed by multiple clients. The repository is the only location that stores the complete history of the project.

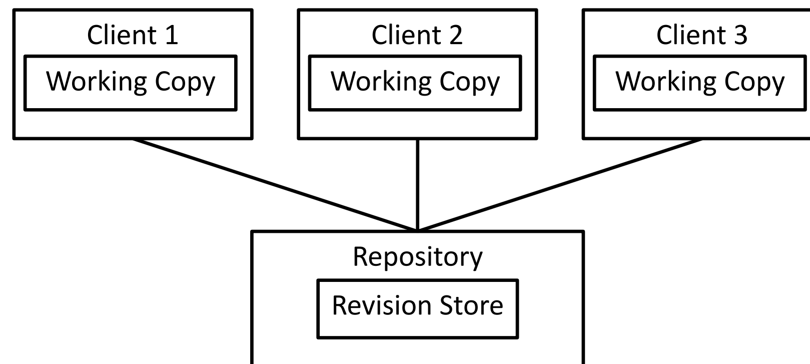

Figure 2.1: A simplified view of Subversion's architecture

A single central repository has desirable consequences. A central repository acts as a gateway to a project's history. It can thus be used to regulate access to the history, for example, by granting permission to access history. As such, a central repository can play a role in the implementation of security policies.

The limitation to one central repository also results in undesirable consequences. Storing the complete history in a single place makes the central repository a bottleneck. Operations that need access to a project's history, such as finding the author who last changed a file, depend on access to the history. In case of remote repositories, such operations are limited by network access bandwidth and can thus require a noticeable amount of time to complete. Furthermore, several operations are disabled when the repository cannot be reached. For example, it is not possible to write changes to the history without a connection to the repository.

Git and Mercurial Git and Mercurial are based on very similar architectures. Both systems are based on a peer-to-peer model. As illustrated in fig. 2.2, the distinction of client and repository that can be found in Subversion does not exist. Instead, every peer has its own working copy as well as replication of the project's history. In DVCSs, peers are commonly named repositories.

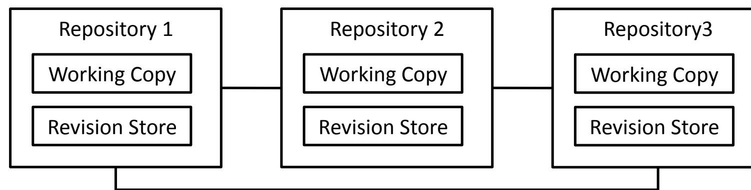

Figure 2.2: Git and Mercurial unify repositories and clients

The absence of a central repository has desirable consequences. The lack of a central authority encourages distributed development. This can be beneficial in open source scenarios, where it allows developers to version the history of their personal changes to a project without requiring write access to public repositories of this project. Furthermore, history replication increases availability and performance. Local history storage allows history-reliant operations to perform without network access and thus faster.

History distribution can also have undesirable consequences. Without a central repository, no single gateway to a project's history, and as of such no single point to implement access control exists. The distinction of local and remote repositories furthermore increases the complexity of the required tool-set, which has to offer additional operations to synchronize repositories.

In summary, the weight of the beneficial and undesirable properties of centralized as well as decentralized repositories depends on concrete use cases. A central repository can be beneficial if access control is of concern, but can at the same time be undesirable, if no central authority for a project exists, and diverging development is encouraged.

2.2.2 History Models

This section analyzes how the chosen VCSs model the history of changes made in working copies. History models can be categorized using certain aspects [CW98]. History models can be *extensional* or *intensional*. Extensional history models expose a set of objects that they version. An extensional history might for example be $h = (v_1, v_2, v_3)$, with v_n being variations of the same object. Intensional history models instead provide means to issue parameterized history queries. An intensional history model might for example expose an interface `getRevision(OperatingSystem, Database, Locale)`. The three chosen systems provide extensional versioning.

Moreover, history models can be *state-* or *change based*. While state-based systems expose the history of a project as revisions of the versioned artifacts as they existed at various points in time, change-based approaches expose the changes made to versioned artifacts. All three VCSs are state-based.

Many state-based history models expose history as a Directed Acyclic Graph (DAG) of revisions [CW98], as illustrated in fig. 2.3. A **revision** consists of a snapshot and meta data. A **snapshot** is an immutable copy of the versioned file hierarchy. The DAG is formed by the **parent** relation that connects child to parent revisions. The snapshot of a revision is assumed to be a merged and modified copy of its parent revisions' snapshots. Concrete state-based history model may limit the maximum number of parents. A revision's meta data may include information such as author of changes, time of commit, or a comment describing the changes.

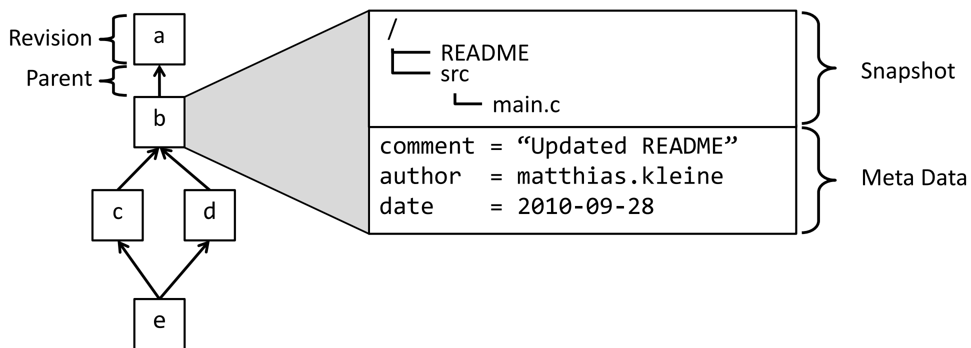

Figure 2.3: Representing history as a directed acyclic graph of snapshots

A history model may be able to represent that branching development was merged back together. Figure 2.3 shows the representation of a merge in a DAG-based history model, revisions *c* and *d* are merged into revision *e*. In state-based history models, merging two revisions is often performed by finding a common ancestor revision and merging the changes that were made in relation to this common ancestor. Merging of files can be performed by an external tool, such as a diff3 tool [KKP07].

DAG-based history models have only limited support for representing operations that act on changes [O'S09]. This includes transferring single changes across branches, so called *cherry picking*. Figure 2.4 shows an example for cherry picking. The original history graph is shown on the left. In this example, the developer wants to extend revision *x* with the changes introduced by revision *b*, without introducing the changes made by *a*. As

seen in the middle history graph, merging x with b is not a solution as it introduces the changes made by a. Alternatively, the developer can copy the changes made by b into a new revision b'. The resulting history graph is shown on the right. The fact that both b and b' are the result of the same changes is not captured in the history model. It can thus not make use of this knowledge, for example when comparing or merging revisions.

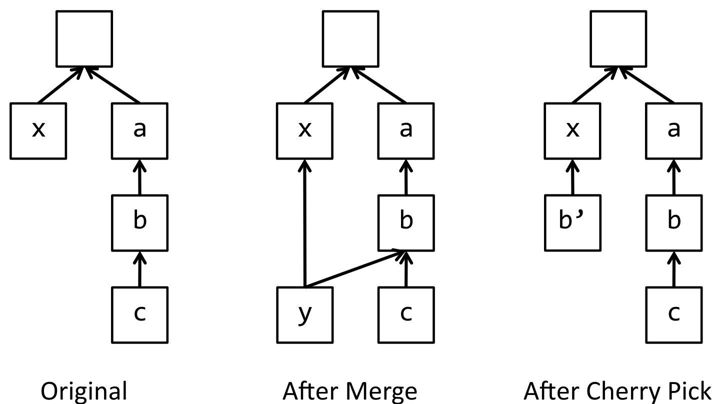

Figure 2.4: Cherry picking the changes made by b onto x

Finally, the history model that is exposed to the user must be distinguished from the storage model employed by the VCS. In fact, all three VCSs expose a variation of the state-based approach to the user but use a change-based approach underneath to reduce storage requirements.

Subversion Subversion has a linear state-based history model. The parent relation is not modeled explicitly. Instead, revisions are numbered sequentially. Revisions contain a snapshot of the complete directory hierarchy. Thus, by writing changes to one or more files back to the repository, a new revision of the complete directory hierarchy is created. If a file is changed by a revision, this revision is said to be *operative* for this file. Clients can request to read a specific revision of files stored in the repository. In addition to a directory hierarchy snapshot, revisions also contain meta data, so called revision properties. Revision properties are exposed as key-value lists and are used to store information, such as author of change. Apart from revisions, individual files can be associated with meta data in the form of key-value lists, so called properties. These are used to indicate information such as line ending convention or merge history.

Figure 2.5 shows an abstract example for Subversion's revisions. The example shows three sequential revisions next to each other. Each revision is visualized as a box that contains revision identifier and commit message. Below, the directory hierarchy is listed and the contents of the file "src/main.c" that exists in all three revisions is shown. As can be seen, the second revision adds the file "util.h" and the third revision makes use of this new file in "main.c".

Revision 1 Initial Commit	Revision 2 Added util.h	Revision 3 Using util.h
trunk README src main.c	trunk README src main.c util.h	trunk README src main.c util.h
src/main.c	src/main.c	src/main.c

int main ()
{
return 0;
}
	int main ()
	{
	return 0;
	}
		#include "util.h"
		int main ()
		{
		useUtil();
		return 0;
		}

Figure 2.5: Example Subversion revisions visualized

Subversion's linear history model cannot represent branching of revisions. Thus, changes that are made within a working copy can only be committed to the repository once they have been merged with the changes from the most recent revision. This requires working copies to store their relation to the history.

Subversion's working copies track their relation to the history by associating each file with the revision that it is based on. This prevents clients from accidentally committing over changes made by others. When a part of the working copy is restored to a revision from its repository, the working copy marks the local files as being based on that revision. Once changes are to be transferred to the repository, Subversion can determine whether the user is trying to overwrite files that were changed in the repository since the working copy was last updated. If this situation occurs, the user

must first get the current revision of the affected files from the repository and merge the remote changes with the local changes. Only then can the changes be transferred to the repository.

The following example illustrates the operation of working copies in Subversion. Given are two developers who work on the repository shown in fig. 2.5. Each developer has his own working copy. The status of the working copies is shown in table 2.1. In this example, both working copies are initially updated to revision 3. Next, both developers modify "src/main.c". As a result, files in both working copies are based on revision 3 but have local changes. Next, developer 1 commits his changes and thus creates revision 4. The snapshot of the directory hierarchy associated with revision 4 is exactly the snapshot that developer 1 had in his working copy. At this point, developer 2 still has a working copy that is based on revision 3 and that has local changes. As he has changed "src/main.c" and as this file has also been changed in the repository, he cannot yet commit his changes. Instead he updates his working copy to revision 4, thus merging the changes he made with that of revision 4. By updating his working copy he discards the information that his changes were originally developed against revision 2. Once the changes are merged, the linearized history can be committed, creating revision 5.

Table 2.1: Example for status of working copies

Action	Working Copy 1	Working Copy 2
	Rev 3	Rev 3
Change working copy	Rev 3 + Changes	Rev 3 + Changes
Developer 1 commits changes	Rev 4	
Developer 2 merges changes		Rev 4 + Changes
Developer 2 commits changes		Rev 5

In summary, Subversion provides a linear state-based history model. Revisions are not ordered by an explicit parent relation, but are addressed by sequential numbers. As of such, Subversion cannot represent branching development as revisions with multiple child revisions. Correspondingly, developers can transfer changes to the repository only after merging them with the latest revision on the repository. Unmerged changes are not represented in the history.

Mercurial and Git Git and Mercurial use non-linear state-based history models. The need for a non-linear history model arises from the system's distributed nature. DVCSs's history models must be able to represent that revisions were created in parallel and are thus based on the identical revision.

The need for a non-linear history model becomes apparent when transferring the example shown in table 2.1 to Git or Mercurial. Again, two developers start out with separate working copies that are both based on the same revision. This time, both developers can modify their working copy and commit their changes to their own repositories without affecting the other developer's repository. If that happens, the two repositories will contain two different revisions that are both based on the same shared revision. This is not possible with a linear history model as that used by Subversion. In Subversion the fact that changes were performed in parallel is discarded and history is linearized.

Both systems implement a history model based on a DAG of snapshots. In contrast to Subversion, revisions are not forced to be ordered linearly. Both systems thus have intrinsic support for branching development. Merging of branches is performed by creating revisions with multiple parent revisions and merging the snapshots.

Distributed history models require revisions to be identifiable across repositories. For example, serializing revisions across repositories relies on being able to identify revisions across repositories. As distributed development requires a non-linear history model, revisions cannot be identified across repositories as single sequential numbers. A version identification scheme that relies on a central authority that issues revision identifiers is undesirable in distributed versioning. Thus, each repository must be able to generate globally unique revision identifiers. This can be achieved by addressing revisions using their content, i. e., associated file snapshots, meta data, and parent revisions. Both VCSs achieve this goal using cryptographic hashes across a revision's content to generate revision identifiers. This so called compare-by-hash strategy ensures that identical revisions that are stored in different repositories will generate the same identifier and at the same time reduces the possibility of using the same identifier for different revisions [Bla06].

In summary, Git and Mercurial provide a DAG based history model. Both systems can thus represent the history of branching and merging development. They do not require developers to merge changes before

committing. As a consequence of the non-linear history model, revisions must be assigned globally unique identifiers.

Differences of Git and Mercurial Git and Mercurial provide different implementations of a directed acyclic graph snapshot model. Up to now, the history models of Git and Mercurial were explained only at an abstract level. This section describes how the concrete history models deviate from the abstract model.

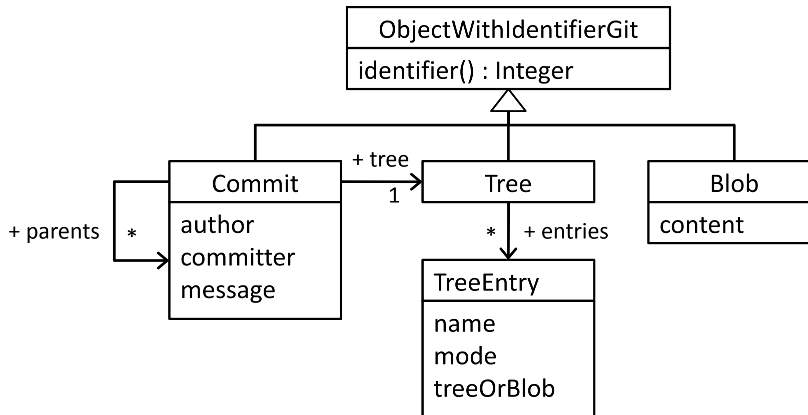

Figure 2.6: A simplified view of Git's object model

Git's history model has three kinds of objects: Commits, trees, and blobs. Figure 2.6 shows a simplified visualization of these kinds of objects. Figure 2.9 illustrates an instance of this model. All three kinds of objects are addressed using an identifier that is generated from their content. A blob consists of arbitrary binary data. A tree represents a file system hierarchy. It mainly consists of a mapping of names to either blobs or trees, each referred to using their identifier. A commit corresponds to a revision. It consists of a tree, a list of parent commits, both again referred to using their identifiers, and various meta data such as commit message or author of change.

Mercurial's history model has three kinds of objects, file contents, manifest, and changeset, as illustrated in fig. 2.7. A manifest is a mapping of path names to file contents. A changeset refers to a manifest and contains additional meta data, such as author and commit message. Mercurial uses a generic DAG-based model to represent history for any of these objects. As of such, revision graphs exist for all three kinds of objects. Objects do not

directly reference other objects, but instead refer to revisions of objects. For example, a manifest does not directly refer to a file contents but instead refers to a revision of a file contents.

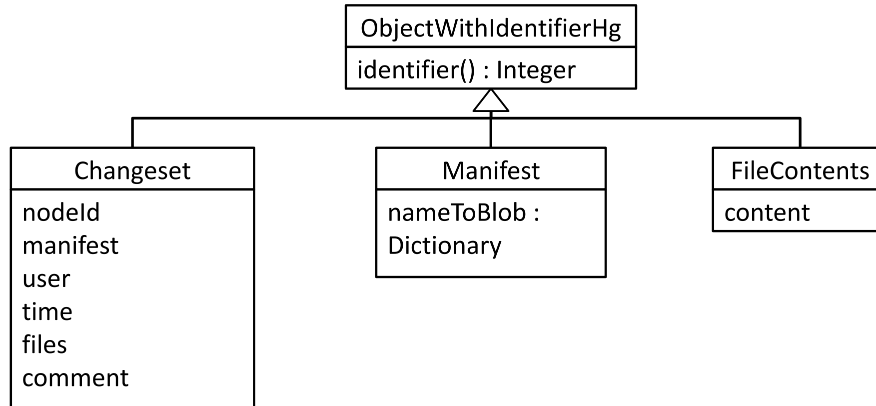

Figure 2.7: A simplified view of Mercurial's history model

The generic history model employed by Mercurial is based on the revlog concept [Mac06]. Revlogs are an efficient implementation of a DAG-based history model for files. As illustrated in fig. 2.8, vertices correspond to snapshots of a file's contents and can be referred to using globally unique identifiers. Edges indicate the parent relationship. Each Mercurial repository has one revlog for changesets, one revlog for manifests, and revlogs for versioned files.

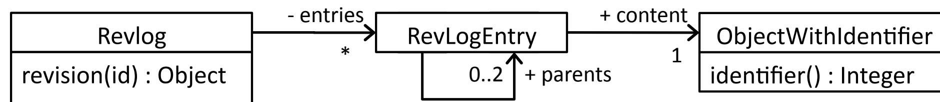

Figure 2.8: A simplified model of the revlog abstraction used in Mercurial

Git's and Mercurial's history models differ in the abstractions that they use to describe history. As illustrated in fig. 2.9, Git stores history only on the level of revisions, whereas Mercurial stores history on the levels of files, file hierarchies, and revisions. The bold arrows indicate parent relations. As a consequence of this, Mercurial's history model has inherent support for capturing copy or move operations on files. If a file is moved or renamed it is still versioned in the same revlog. This allows tracking back changes across renames. Git relies on heuristics to do so but has the benefit

of a simpler model. Heuristics are needed in both systems, if movement of content at a granularity smaller than files is to be detected. In conclusion, both models have small differences, but are very similar when contrasted with Subversion's history model.

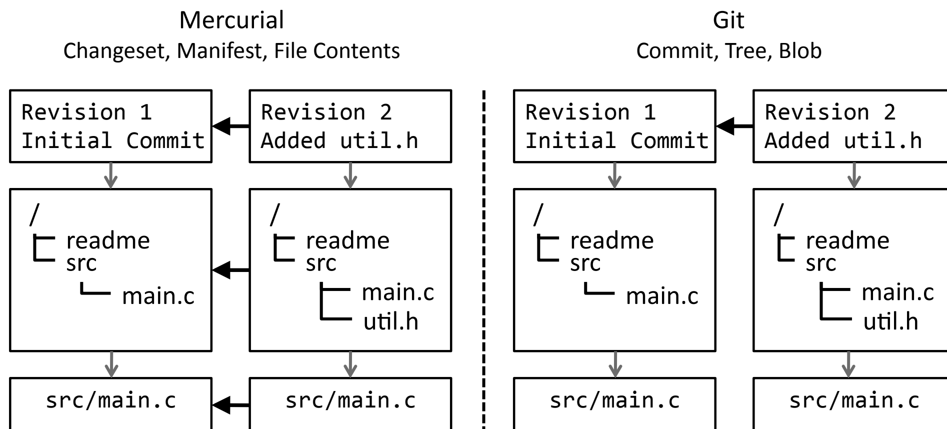

Figure 2.9: Exemplary comparison of Mercurial's and Git's history models

History Models Summarized The history models of the chosen VCSs fall into two categories. Subversion relies on a linear history model. Even though changes do happen in parallel in Subversion in multiple working copies, committing them to the repository discards the information that changes were performed in parallel.

In contrast, Git and Mercurial provide a DAG-based history model. The history models can represent that changes were made in parallel, a prerequisite for DVCSs. The history models of Git and Mercurial apply a nearly identical approach to solve this problem. Revisions have unique identifiers and can thus be addressed and transferred across repositories.

2.2.3 Branching Models

The three systems provide different means for identifying branches. Git and Mercurial provide branching models that are separated from their history models. This separation does not exist in Subversion.

Subversion In Subversion, branches are represented as well as identified by directories of the versioned file hierarchy. Each revision stores the state of all branches. Branches are created by copying directories, and merged by merging the changes of one directory to another one. Representing branches on top of the history model is necessary, as Subversion's history model cannot represent branching revisions. Subversion does as of such not clearly separate the notions of history- and branching model.

The naming and organization of branch directories is governed by best practices. By convention, the top-level directory that is being versioned has the three subdirectories "trunk", "branches", and "tags". This convention assumes that a single central branch is shared by all developers. This is the "trunk". If other branches are required, they are created by copying the directory of an existing branch to a new subdirectory of "branches". Copying is performed using Subversion's "copy" command. This command creates a new revision with meta data that indicates where newly added files were copied from. This information is later on used to establish common ancestors during merging.

Subversion allows branches to be merged. As a first step of merging branch "branches/a" into branch "branches/b", Subversion identifies all operative revisions of "branches/a" that have not been merged into "branches/b". In order to identify these revisions, Subversion keeps track of past merges. Each directory can have a so called mergeinfo meta data property. This property stores a list of branches and revisions that have been merged. Thus, the mergeinfo property can represent that only single revisions were merged across branches, so called cherry picking. The actual implementation of mergeinfo is more complex and must deal with various exceptions, such as partial merging of branches[4].

Table 2.2 illustrates Subversion's mergeinfo with an example. The repository in this scenario contains two branches *a* and *b*. The table lists repository actions and their result on the operative revisions and mergeinfo of both branches. Up to action 4, both branches are created and diverging development was performed. Branch *a* was modified in revisions 1 and 3. Branch *b* was modified in revisions 2 and 4. Action 5 merges *a* into *b*. The mergeinfo of *b* is updated to reflect this. The mergeinfo is extended to cover non-operative revisions. This is done to keep the mergeinfo easier to read

[4] `http://www.collab.net/community/subversion/articles/merge-info.html` – last checked 20.12.2010

Table 2.2: Subversion's mergeinfo illustrated

Action	Operative Revisions		Mergeinfo	
	a	b	a	b
1. Init Repo	(1)			
2. Create *b*	(1)	(2)		
3. Modify *a*	(1, 3)	(2)		
4. Modify *b*	(1, 3)	(2, 4)		
5. Merge *a* into *b*	(1, 3)	(2, 4, 5)		a : 2–4
6. Modify *a*	(1, 3, 6)	(2, 4, 5)		a : 2–4
7. Merge *a* into *b*	(1, 3, 6)	(2, 4, 5, 7)		a : 2–6

after consecutive merges. For example, after modifying *a* in action 6 and merging these changes to *b* in action 7, *b*'s mergeinfo is set to includes *a*'s revisions 2–6.

The result of a merge is a single revision that integrates the changes from merged revisions into the files on the destination branch. A merge revision cannot be distinguished from a revision that manually added all changes made in the source branch. It is therefore not possible to merge the destination branch back to the source branch by identifying unmerged revisions, as outlined above. Subversion does thus not support repeated bi-directional merges. For example, if branch "branches/test" is created as a copy from "trunk", changes on "trunk" can be repeatedly merged into "branches/test", but changes on "branches/test" cannot repeatedly be merged back into "trunk". Instead, Subversion offers a so called "integration merge" that performs a 3-diff merge using the files in both branches and the original revision when the branch was created as a common ancestor. After an integration merge, the source branch is to be discarded.

In summary, branches are implemented in Subversion as directories. The history of merging is stored on a per-directory basis using so called mergeinfo properties. Mergeinfo properties can represent that only single revisions were merged across branches. Subversion can thus represent cherry-picking of changes across branches. Yet, it does not allow repeated bi-directional merges.

Git In Git, each repository maintains a mutable dictionary of branch names to revision identifiers. For example, a repository might store that

branch "main" corresponds to revision "715" and that "test" corresponds to revision "825". We name this style of branch-representation label-based branching. In addition to its own branch dictionary, a repository stores copies of the dictionaries of other repositories. These are exposed to the user by prefixing branch names with the name of the repository that they are owned by. For example, "origin/main" is the branch "main" on the repository "origin". These local branches that correspond to remote branches are called tracking branches.

Git additionally allows each local branch to have a so-called upstream branch. If a branch has its upstream branch set, Git's commands provide information on how the two branches evolved in relation to each other. For example, the local branch "master" can have its upstream branch set to branch "master" on repository "origin". If a new commit is made on both branches, Git's status command will report "Your branch and 'origin/master' have diverged, and have 1 and 1 different commit(s) each, respectively." Git's "branch" command can report similar information for all branches contained in a repository.

Upstream branches are helpful when interacting with more than one repository. This is common in open source projects where central repositories can only be written by trusted developers. Untrusted developers must use their own repositories to share changes and can request trusted developers to use their changes.

Figure 2.10 illustrates a concrete example. On the left it shows a public repository that can be read by anyone. This repository has two branches that contain a stable as well as an untested version of the product. The public repository can only be written by selected developers. The developer "Contributor" works on a new feature but cannot write to the public repository. He must employ a mediator to publish his changes. He keeps a local branch "publish" that always points to a commit that he wants to be published and asks a developer "Integrator" to watch this branch.

The developer "Integrator" is amongst those who can write to the public repository. His role is to integrate changes made by others. His repository contains branches that correspond to the branches on the public repository, as well as a branch "contrib" that has "publish" on "Contributor"'s repository as its upstream branch. Thus, Git will notify "Integrator" whenever this branch changes. It will furthermore simplify merging changes from an upstream branch. Thus, Git helps keeping track of branches that correspond to each other and simplifies interactions between them.

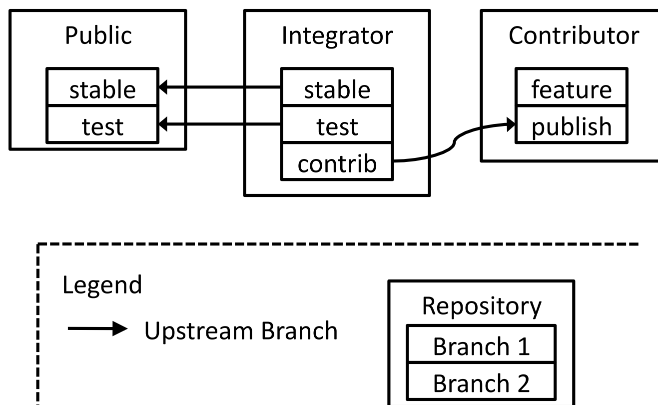

Figure 2.10: An example of upstream branches in Git

In summary, Git provides a label-based branching representation. A branch is a label that points to a single revision. In Git, repositories cache branches of remote repositories. A branch can furthermore have an upstream branch, to which it is compared and merged with by default.

Mercurial Mercurial's revisions store the name of the branch that they belong to as part of their meta data. For example, a revision may store that it belongs to branch "default", whereas another may store that it belongs to branch "testing". As such, Mercurial does by default not have an explicit reification of a branch's current revision. Multiple childless revisions with the same branch-name may exist. A branch name can still be used to identify a revision. Given all revisions that belong to a branch, the revision that was added most recently to the repository is said to be the current revision of that branch. As the current revision of a branch depends on the order that revisions were added to a repository, two repositories can report different current revisions for a branch, even if they contain exactly the same revisions.

Mercurial's lack of an explicit reification of a branch's current revision forces developers to perform certain tasks outside of the VCS. For example, it can be desirable to investigate revisions from another repository without changing information about local branches. As Mercurial's branching model cannot separate transferring revisions to a repository from changing the current revisions of branches, it is impossible to refer to both the remote as well as the local current revisions of a branch by name.

Mercurial can be extended to support a branching model similar to that of Git with the help of the "bookmark" extension. Bookmarks also offer a name to revision mapping. Yet, unlike Git's branches, bookmarks are not owned by repositories. In Git each repository has its own namespace for branch names, so that a branch named "main" can have a different version than a branch with the identical name on a different repository. Mercurial's bookmarks share one global namespace. Transferring revisions from one repository to another one automatically changes bookmarks with the same names on the destination repository. If one wants bookmarks to be visible to other repositories but at the same time wants bookmarks with identical names to point to different revisions, one has to enforce this manually.

In summary, Mercurial provides a revision-based branching model. Each revision stores the name of the one branch that it belongs to. This branching model does thus not have an explicit reification of a branches' current revision. A branching model similar to that of Git exists as an extension of Mercurial, so called bookmarks.

2.2.4 Summary

This section analyzed and compared the three VCSs, with special focus on the concepts of repositories, history-, and branching models. The analyzed VCSs exhibit differences in all of these concepts. Two kinds of repository organizations exist, centralized and distributed. Two kinds of history models exist, linear history and DAG-based history. Three kinds of branching models exist, manual directory-based branching, label-based branching, and revision-based branching.

Given the analysis of the three systems, the next step is to extract requirements that are the basis for an abstraction that can be implemented in all of these three systems.

2.3 Requirements

Pur should provide an abstraction that is sufficiently rich to be the basis for client programs that perform consistently across VCS. This can be guaranteed by satisfying various requirements, foremost providing an abstraction that has sufficiently specified semantics and that does not expose specifics

of supported VCSs. This sections presents and discusses these and other requirements.

2.3.1 Provide Rich Semantics

In order to be the basis of complex version control client programs, Pur must have sufficiently rich and specified semantics. For example, it is desirable to provide a construct for identifying branches that behaves uniformly across VCSs. As a consequence of requiring rich semantics, providing abstractions that deviate from concrete VCSs is preferable to providing abstractions with no specified semantics. For example, it is preferable to provide a branching model that for Mercurial must be implemented using the non-standard bookmarks to providing a branching model that behaves differently across VCSs.

2.3.2 Version Control System-agnostic Interface

Pur must not expose details of supported VCSs that are not relevant to other VCSs. For example, although branching development is implemented using directories in Subversion, this implementation details must not be exposed by Pur, as it is irrelevant to Git and Mercurial. By not exposing specifics of underlying VCSs, client programs are guaranteed to work against any supported system.

2.3.3 Minimal Interface

Pur should not cover aspects that are relevant only to particular client programs, such as convenience methods that can be reconstructed from other methods, nor should it cover aspects that are relevant only to implementors of Pur for concrete VCSs, such as concepts of version identifiers or the distinction of local and remote repositories. By making minimality one of the design goals of Pur, an overly complex architecture and resulting drawbacks can be prevented. This is desirable, as superfluous complexity is likely to hamper the adaption of Pur.

2.3.4 State-based Non-linear History Model

The analyzed VCSs all rely on a state-based history model. The abstraction thus must also provide a state-based history model. In order to support Git's and Mercurial's non-linear history, the model must be able to represent arbitrary directed acyclic graphs.

2.3.5 Consistent Branching Model

It is desirable to have a branching abstraction in Pur whose semantics are specified to a degree that allows branches to work consistently across various VCS back-ends. In particular, calculating a branch's current revision should be consistent across back-ends. Furthermore, distinguishing between local and remote branches should be handled consistently.

2.4 Summary

This section provided the background for an abstraction over version control concepts. Three VCSs were analyzed for common concepts and resulting requirements for an abstraction were established. The following sections describe and evaluate how Pur addresses these requirements.

3 Pur—An Abstraction for Version Control

This chapter introduces Pur (IPA [puːɐ̯]), an abstraction over version control constructs. Pur abstracts over the concepts of the VCSs Git, Mercurial, and Subversion. Client programs can interact with any of these VCSs through Pur, without requiring knowledge of the VCS's specifics. Pur is designed to address the requirements identified in the previous chapter.

Pur is specified as a set of object oriented interfaces. This set consists of interfaces for versioned objects, history of versioned objects, branching, and repositories. This chapter introduces these interfaces in this order. The specification of the interfaces is concluded with concrete examples of Pur, which aim to give a practical understanding.

Figure 3.1 illustrates the interfaces of Pur using a UML-like notation. Role-names do not indicate that referencing objects must store references as fields/instance variables. Instead, role-names indicate that referencing objects must provide access to the referenced object via message send. Furthermore, all operations provided by the interfaces can fail. It is the caller's responsibility to deal with failures.

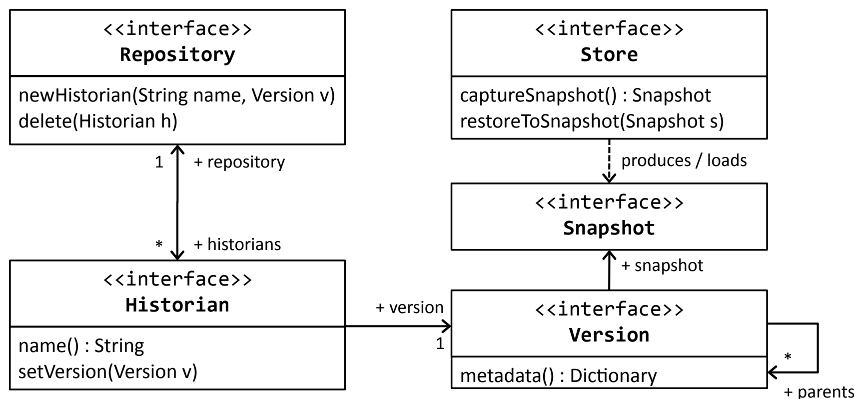

Figure 3.1: Interfaces for the objects being versioned by Pur

3.1 Stores and Snapshots

A store is a mutable object that is to be versioned. A snapshot is an immutable deep copy of a store. A store can capture its current state as a snapshot and can restore its state to that captured in a snapshot by loading it.

3.2 Versions

Pur represents history as a directed acyclic graph of snapshots represented by versions. A version consists of a snapshot, a list of its parent versions, and meta data, such as commit message, author, or time of creation. The parents of a given version v are said to have v as its child. The terms "ancestors" and "descendants" are used to refer to the transitive closure of the parent (respectively child) relation.

3.3 Historians and Repositories

Pur provides label-based branching through historians. A historian has a name, a version, and can be requested to be set to a different version. A historian is owned by a repository. A repository owns a set of historians that it exposes. It allows creating new historians and deleting existing ones.

Historians are not named "labels" to underline the fact that a historian is not merely a name and a version but also encapsulates access to this version. Historians are furthermore not named "branches" to avoid confusion what constitutes a branch across VCSs.

3.4 Pur by Example

So far only an abstract description of Pur was given. The following section aims to provide a more practical understanding of Pur with the help of a few example scenarios. First, practical examples of stores and snapshots are given. Next, visual examples illustrate the history model of Pur as well as the interaction with it.

3.4.1 Stores and Snapshots

Stores reify the objects being versioned. For this example we assume a file based environment. Thus a store corresponds to the versioned directory hierarchy and a snapshot to an immutable copy of it. We visualize snapshots as boxes with text that corresponds to the contents of the files, as seen in fig. 3.2. This visualization does not show any information about how the snapshot is synchronized, diffed, or merged. It is nevertheless a useful visualization to show differences between snapshots.

```
class Main = ()()
```

```
class Main = ()(
  foo = ())
```

Figure 3.2: Visualization of snapshots

3.4.2 Versions

Figure 3.3 shows a history graph of versions with two consecutive versions shown in detail. The topmost version labeled *a* has no parents, a snapshot of an empty store, and no meta data. The version *b* has *a* as its only parent. Its snapshot contains the source code of the class "Main". The only child of this version is version *c*. The child has a different snapshot that contains an additional method, as is also indicated by the version's comment.

Versions can represent branching development. The versions *d* and *e* the same parent version. The changes of both versions are merged back into one version *f*.

3.4.3 Historians

When developing branches it becomes desirable to assign names to diverging branches. Pur addresses this need with **historians**. Historians provide label-based branching and thus allow identifying versions that are currently being worked on. For the sake of simplicity, all historians in this example are assumed to be owned by a single repository. Figure 3.4 shows a version graph at different points in time. Part 1 shows the initial version graph. Only one version is shown, the rest of the version graph being left out. There is exactly one historian, named "share" that has this one version as its version. In this example, team members agreed that naming a historian

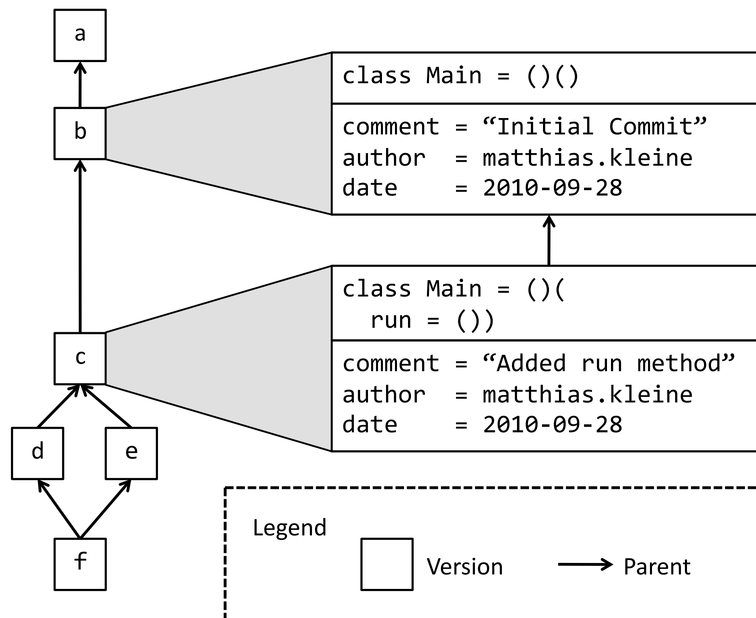

Figure 3.3: Visualization of a history graph. Two versions shown in detail

"share" indicates that the historian's version is sufficiently stable to be used by all developers.

At this point in time a developer performs a large refactoring that changes interfaces that are used by other developers. It is desirable to record the refactoring as a series of versions, as it allows dividing the refactoring into logically separated steps. The intermediate versions contain incomplete refactorings that the developer does not want to share with other developers. The changes thus cannot be made using the "share" historian. Instead the developer creates a new historian named "refactor" that initially has the identical version as "share", as can be seen in part 2.

As shown in part 3, the developer next starts working on the refactoring, thus creates new versions and advances the "refactor" historian. The developer continues to create new versions using the "refactor" historian. Simultaneously, other developers advance the "share" historian by creating new versions. The resulting state is shown in part 4. Finally, the developer decides that he wants to share his changes with other developers and thus merges them with the version of the "shared" historian. He furthermore

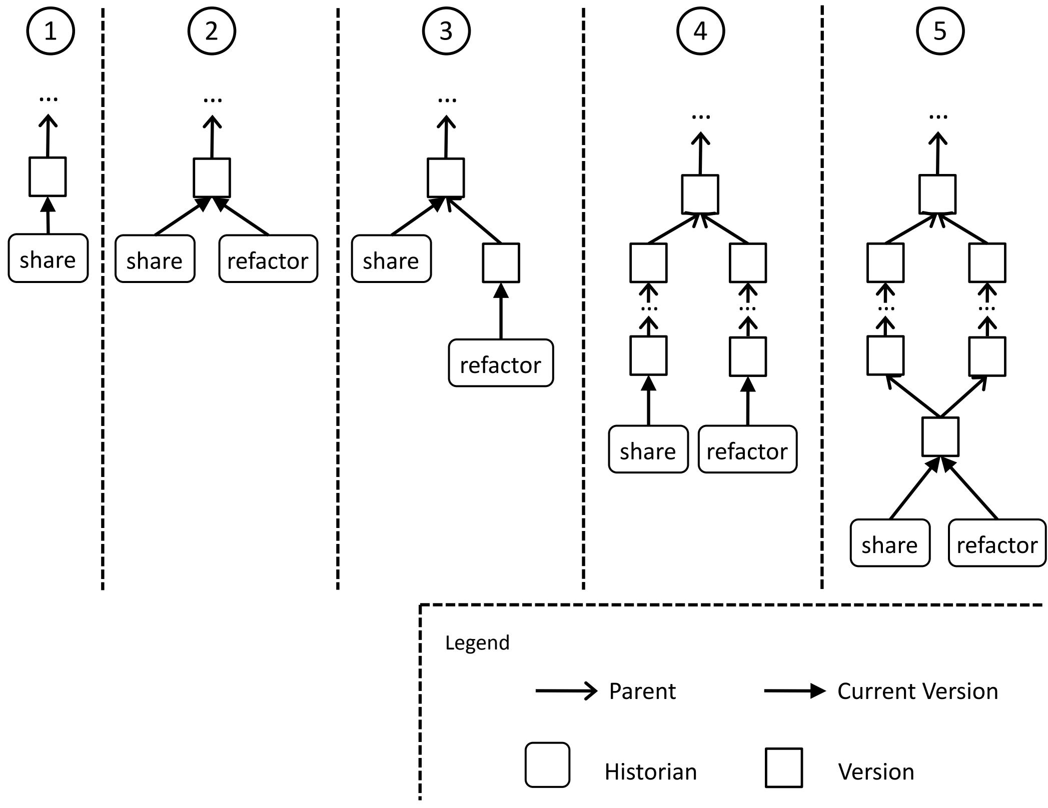

Figure 3.4: Historians create new versions

sets the "shared" historian to this newly created version. This can be seen in part 5.

3.5 Summary

This section introduced Pur, an abstraction over version control concepts. Pur is specified as a set of object oriented interfaces that cover various version control concepts. Based on these interfaces, concrete implementations of Pur as well as client programs that make use of Pur can be built. Both kinds of implementations form the basis for the following sections.

4 Implementing Pur for Concrete Back-ends

This chapter describes Pur implementations for Git and Mercurial in the Newspeak programming platform [BAB+08]. It thus validates the implementability of Pur. The Newspeak programming platform encompasses the Newspeak programming language, an IDE, and various libraries. The implementation of Pur within Newspeak serves as a basis for the version control application PNS that is described in chapter 5.

This chapter is structured as follows. Implementation guidelines are discussed that were followed during the implementation to achieve maintainable and extensible code. Next, the implementation of the concrete back-ends is explained. Currently, back-ends for Git and Mercurial are implemented. An implementation strategy for Subversion is outlined.

4.1 Abstract Implementation

It is desirable to make the implementation of Pur extensible and maintainable. These goals can be achieved by applying software engineering's best practices, such as modular design. This section analyzes possible design choices and explains how these properties can be achieved by implementing Pur as a framework.

4.1.1 Implementation-specific Requirements

Pur does not include concepts that are relevant only to either VCSs or client programs. Instead Pur addresses concepts that are shared by both. For example, Pur does not include version identifiers, nor does it include diffing algorithms. Concrete Pur implementations must provide these concepts both on top of and below Pur. Some of these concepts are relevant across VCSs, respectively, VCS clients. For example, a construct for version identifiers can be found in any of the supported VCSs. Diffing algorithms are relevant to multiple VCS clients.

Implementing these shared concepts separately for each VCS and client program would result in repeated implementation effort. The implementation effort can be kept low by implementing Pur as a framework that captures and orchestrates shared concepts both on top of and below Pur. Concrete VCSs can specialize this framework and thus can rely on the shared implementation provided by the framework. The following section explains the design of this framework.

4.1.2 Architecture Overview

Pur is implemented as a framework that is provided by a set of abstract classes that implement parts of the interfaces described in chapter 3. These abstract classes can again be divided into a generic implementation of Pur on the one hand, and specializations for repositories that are accessed via working copies and repositories that are accessed through the working copy of another repository. We name the former local repositories and the latter remote repositories. The benefits of this distinction are explained in section 4.1.3.

This section first describes the generic implementation and later on the specializations for local and remote repositories. Figure 4.1 depicts the classes provided by the generic part of the framework. The notation is based on UML, the difference being that methods shown in gray must be implemented by subclasses. Additionally, slots as well as methods are expected to be accessible via message sends. Thus, a slot can be a valid implementation of a message required by an interface. The names of the classes are identical to those of the interfaces. If not stated otherwise, this section refers to classes.

The class `Version` serves as a base for all implementations of the `Version` interface. It provides common methods, such as `compareTo`, which calculates the differences to another version, or `versionsIncomingFrom`, which takes a version v and returns all ancestors of v that are not part of the receiver's history. These common methods require the concrete subclasses to implement the methods marked as missing. E. g., `compareTo` must first calculate a common ancestor of the two versions being compared which itself relies on `parents` being implemented.

The abstract class `Historian` is the base class for all historians. A historian stores its owning repository and its name as fields. The abstract class `Repository` does not provide any implementation at all. Instead it merely defines the required interface.

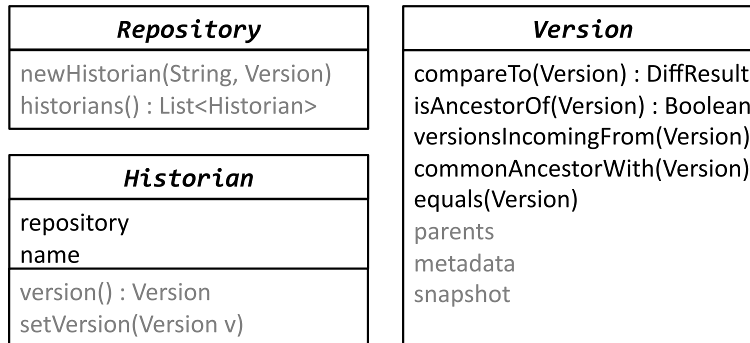

Figure 4.1: Abstract classes provided by framework; gray indicates methods that must be implemented by subclasses

4.1.3 Local and Remote Repositories

Pur distinguishes local and remote repositories. Only local repositories expose a working copy. Using this working copy, local repositories allow users to read and write versions, including their snapshots. In contrast, remote repositories expose only partial information about the versions that they store, including the versions' identifiers, but not their snapshots. In order to access the snapshot of a version stored in a remote repository, this version first must be transferred to a local repository. As found in section 2.2.2, repositories assign unique identifiers to versions. Using these identifiers, versions can be transferred across repositories.

Local Repositories The classes provided by the specialization for local repositories are shown in fig. 4.2. The class `LocalRepository` is the base class of all local repository implementations. As its superclass `Repository`, it does not provide any concrete implementations. Instead it extends the interface that has to be provided by concrete implementations. It adds a method to retrieve a version from the back-end using a back-end specific identifier, and two further methods to transfer versions to and from other repositories. Both of these act on version identifiers instead of on versions.

The class `LocalRepositoryVersion` captures common concepts for versions that are accessed through a local repository. A local repository version stores its repository, a version identifier and meta data as fields. Specializations of `LocalRepositoryVersions` can assume the presence of these fields and must provide access to snapshots and version identifiers of parent versions. Operations for accessing parent versions or comparing versions

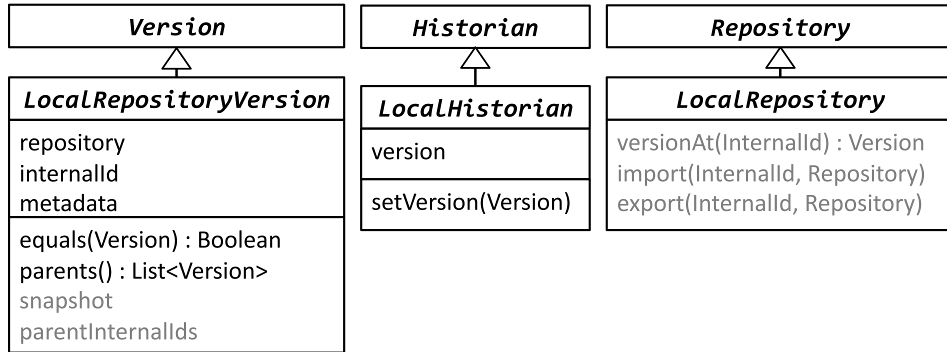

Figure 4.2: Specializations for local repositories

for equality using version identifiers can then be implemented once in `LocalRepositoryVersions` and thus prevent duplicate implementation.

The class `LocalHistorian` is the base class for all local historians. A local historian stores its version as a field that contains a `LocalRepositoryVersion`. `LocalHistorian` does provide an extendable implementation for `setVersion` that stores the version in this field. Subclasses can extend this implementation as to reflect the change in version in the concrete back-end.

Remote repositories Figure 4.3 depicts the specializations of the framework for remote repositories. Remote repositories are represented as instances of `RemoteRepository`. Accessing versions stored in a remote repository requires access to a local repository. A `RemoteRepository` thus always has a `LocalRepository`, to and from which versions can be transferred. This is performed via the import and export methods which make use of the local repository's corresponding methods.

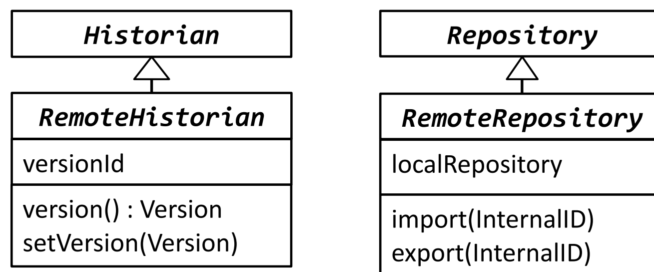

Figure 4.3: Specializations for remote repositories

A `RemoteHistorian` stores only its version identifier. It provides methods to read and write its version which work on `LocalRepositoryVersions`. While reading a version ensures that the version is first transferred to the local repository, setting a version ensures that the new version is transferred to the remote repository.

4.1.4 Extensions

Various extensions to the concepts provided by Pur were implemented. These extensions are not essential to Pur but instead provide features that are relevant only to certain client programs. As such it is desirable to implement them as a layer on top of Pur. This section introduces two extensions: Tracked repositories and upstream historians[1].

Tracked Repositories This extension allows each repository to have a collection of other repositories that it interacts with, so called tracked repositories. Tracked repositories allow developers to keep track of repositories that they interact with and thus eliminate the need for a central authority of repositories.

The use of tracked repositories can be illustrated using the following example. A developer works on an open source project. This project has an official public repository R_p that is controlled by the project's maintainers. In addition to the official public repository, several developers publish their own repositories $R_{d1} \cdots R_{dn}$ that contain code that has not been accepted by the project's maintainers.

The developer in our example only wants to interact with a subset of all of these repositories. No central authority for repositories exists. Instead, developers must maintain the set of repositories that they interact with. This can be accomplished using tracked repositories. Assuming, that R_{dx} contains extensions to the project that are not included in R_p, the developer can store R_p as well as R_{dx} as tracked repositories. The tracked repositories can then be exposed to the developer by a version control application that allows browsing historians and notifies the developers of changes.

Upstream Historians This extension allows each `LocalHistorian` to store a historian that it corresponds to as its upstream historian. The concept of upstream historians corresponds to the concept of upstream branches

[1]The terms "tracked" and "upstream" are taken from the Git terminology.

of Git, see section 2.2.3. As such, upstream historians allow establishing links between historians that are commonly related to each other, in order to allow version control applications to compare and merge such related historians automatically.

For example, upstream historians can be set by default when creating new historians on the basis of existing remote historians. When local and remote historian diverge, a source control application can inform developers about this and offer appropriate actions. For example, the local historian "main" may be created as a clone of the historian "main" on a central repository. If "main" changes on the central repository, the version control application may notify the user of this and offer to reset the local historian to the remote historian's version.

Upstream historians are not required to compare historians. Yet, the number of historians that can potentially be compared grows with the number of tracked repositories and historians. Thus, a way to reduce the number of relevant historian-pairs is required and upstream historians provide one way to address this problem.

4.2 Implementation for Back-Ends

This section describes how Pur was implemented for Git and Mercurial using the previously described framework. It also describes a implementation strategy for Subversion. Throughout this section table 4.1 is used to compare the implementation of actions in Mercurial, Git, and Subversion.

4.2.1 Git

The classes provided by the framework correspond to constructs of Git. Pur's abstractions version, history, and repository are mapped to Git's commit, branch, and repository, respectively. The `LocalRepositoryVersion` maps directly to Git's commit, with Git's commit identifiers serving as Pur's internal identifiers. `LocalHistorian` directly maps to Git's local branches, and `RemoteHistorian` maps to to Git's remote branches.

Setting a historian's version is mapped to pushing the corresponding commit to the branch of the corresponding repository. E. g., setting branch "master" on repository "repo" to the version with identifier "123" executes "git push repo 123:master" within the working copy of the local repository. This works for local as well as for remote historians. Pur's repositories are

Table 4.1: Comparison of corresponding VCS commands

Action[a]	Git	Mercurial	Subversion
Create or Set “main” to ID	push server ID:main	debugpushkey server bookmarks main OLDID ID[b]	propset --revprop --revision 0 historian_main ID
Delete “main”	push server :main	debugpushkey server bookmarks main OLDID `'[c]	propdel historian_main --revprop --revision 0
Import ID	fetch server[d]	pull server --rev ID	n/a
Export ID	n/a[e]	push server -f --rev ID	n/a
Get meta data of ID	log --pretty=format:%H%n %P%n%a%n%n%s -n 1 ID	log --template '{node}\n{parents} \n{author}...' --rev ID:ID	log --revision ID --xml
Prepare for merge with ID	merge --no-commit --strategy=ours ID	--config ui.merge=internal:local merge ID	Internal
Commit working copy with comment “Bugfix”	commit --message=Bugfix	commit --message Bugfix	Internal
Load historian “main” into the working copy	checkout main	update -C main	update --revision `svn propget --revprop --revision 0 historian_main`

[a] If not indicated otherwise, actions work on historian “main” on repository “server”, and version “ID”
[b] where OLDID is main’s previous version, or "" if non-existent
[c] where OLDID is main’s previous version
[d] Single versions can not be requested
[e] Not required, as setting a historian’s version via “push” automatically transfers the versions

mapped to Git' repositories. Importing and exporting versions is mapped to the "fetch" and "push" commands.

The extensions can be implemented directly using Git's constructs, see table 4.2. Git provides constructs to store a list of remote repositories in a repository. This is used to implement tracked repositories. Git furthermore allows each branch to have an upstream branch. This is used to implement upstream historians.

Table 4.2: Implementation of extension commands

Action	Git
Get name and repository of "main's" upstream historian	`for-each-ref --format %(upstream)refs/heads/main`
Set "main's" upstream historian to "other" on "server"	`branch --set-upstream main refs/remotes/server/other`[a]

[a] where the remote branch must first be imported using "fetch server other refs/remotes/other"

4.2.2 Mercurial

The classes provided by the framework correspond to constructs of Mercurial. Versions correspond to changesets, historians to bookmarks, and repositories to repositories. `LocalRepositoryVersions` correspond to Mercurial's changesets, with node ids being version identifiers and parent node ids accordingly.

Historians are implemented using Mercurial's bookmarks. These do by default behave differently than historians: Mercurial's bookmarks share one global namespace. When changesets are pushed to another repository Mercurial checks if these changesets are referenced by any bookmarks. If that is the case, Mercurial will try to update corresponding bookmarks on the destination repository. This is not in conformance with the desired behavior of having one historian namespace per repository.

Local namespaces thus must be simulated on top of bookmarks. This can be done in at least two different ways. First, it is possible to simulate Git's approach of storing historians of both local and remote repositories within a local repository. Second, one can store only the repository's own historians as its bookmarks. Both approaches have drawbacks. The approach

of storing both local and remote historians locally requires encoding the historian's owning repository into a bookmark name. This again would require repositories to have unique identifiers. No obvious solution for this exists.

The approach of storing only local historians can make use of the fact that historians of remote repositories can be read and written without influencing any local bookmarks using Mercurial's "debugpushkey" command. In order to prevent automatic bookmark synchronization, this approach must ensure that the bookmark extension is disabled for commands that transfer revisions. Unlike the first approach, remote historians can only be read if a connection to remote repositories is available. The latter approach was chosen nevertheless, as its implementation does not does not require new concepts, such as repository identifiers, to be introduced into Mercurial.

Repositories are implemented using Mercurial's repositories. The `LocalRepository` relies on having access to the repository's working copy, while the `RemoteRepository` requires only a repository identifier that Mercurial can make use of. `LocalRepository`'s import and export methods have to ensure that the bookmark extension does not automatically update bookmarks in any repository. This can be achieved by temporarily disabling Mercurial's bookmark feature.

The extensions of Pur do not directly correspond to concepts of Mercurial, but must partly be simulated. Mercurial can store a list of remote repositories within a working copy. This is used to implement the tracked repositories. Mercurial does not provide a way to store upstream bookmarks. Upstream historians are thus stored in a custom section of Mercurial's configuration files.

4.2.3 Subversion

A concrete implementation of Pur for Subversion has not been established due to time constraints. This section describes an implementation strategy in order to show the feasibility of such an implementation.

Pur can be implemented for Subversion in various ways. A naive approach would be to adapt Subversion's linear history model. This would not allow changes to be recorded in parallel. As being able to do so is essential in collaborative editing, this is not an adequate approach. Instead, it is possible to use Subversion repositories as storage for arbitrary data and implement non-linear history on top of Subversion's linear history.

Pur's versions can be implemented using Subversion's revisions. A revision does by default not have an explicit reification of parent revisions. This can be added using Subversion's revision properties, see section 2.2.2. Subversion allows arbitrary key-value pairs to be added to revisions as meta data. Thus, a revision may have the meta-data "`parents=3,7`" to indicate its parent revisions.

Subversion's revision properties can also be used to represent historians. Revision properties are mutable. Thus, it is possible to store the state of historians in a single dedicated revision, for example the first revision of the repository. For example, one could store the meta-data "`historians=main:317, testing:512`". Setting historians to other versions is then implemented by changing this meta data.

Repositories correspond to Subversion's repositories. Thus, by default, only a single repository exists. This would make a large part of the abstract implementation irrelevant to Subversion, as there is not distinction between local and remote repositories. A evaluation can only be performed by implementing Pur for Subversion.

Subversion does not provide a command to create a revision whose file hierarchy snapshot corresponds exactly to that of the working copy. Instead, Subversion's `commit` command tries to integrate changes with those made by others. If local and remote changes affect distinct files, they are integrated. Creating a new version therefore requires interacting directly with the Subversion library. The Subversion library provides operations to directly interact with a repository's underlying file system.

4.2.4 Summary

This section showed that Pur can in practice be implemented for the chosen VCSs. The provided implementations rely on a shared abstract implementation in order to reduce code redundancy. Across the implementations, different conceptual mismatches are addressed. One can observe, that Pur maps nearly directly to Git, requires some amount of simulation for Mercurial, and requires a complete history model to be simulated for Subversion. Given these implementations, Pur can be represented in Git, Mercurial, and Subversion.

5 Pur for Newspeak—PNS

The applicability of Pur remains unevaluated without a client application. As part of this research, PNS—Pur for Newspeak—was implemented. PNS is a version control application that is used to version Newspeak code. PNS is layered on top of Pur and can thus store history in Git or Mercurial.

This chapter explains the implementation of PNS. It is structured as follows: First, the implementation of snapshots of Newspeak classes is given. Based on that, two kinds of stores are explained that can load and produce these snapshots. Next, the version control application itself is presented. Finally, an evaluation of PNS is given.

5.1 Snapshots

PNS must be able to capture snapshots of versioned objects, i. e., Newspeak classes. It does so by providing a structured representation of class snapshots, as illustrated in fig. 5.1. This structured representation was chosen in favor of an unstructured text representation, at it allows re-using the same representation for other operations, such as diffing.

PNS provides three kinds of snapshots. A StoreSnapshot is the only snapshot that is directly exposed to Pur. It corresponds to a snapshot of a store and as of such consists of class snapshots, represented by the class `ClassSnapshot`. A class snapshot provides fields for all elements that make up a class.

While methods are captured in custom `MethodSnapshots`, all slots of a class are stored as one plain string. This is due to a limitation of the current form of the Newspeak grammar which does not yet treat comments as meta data, but simply discards them. If slots are interspersed with comments, these comments cannot be attributed to any slot. This is due to change, with slots being handled as meta data that is accessible after parsing.

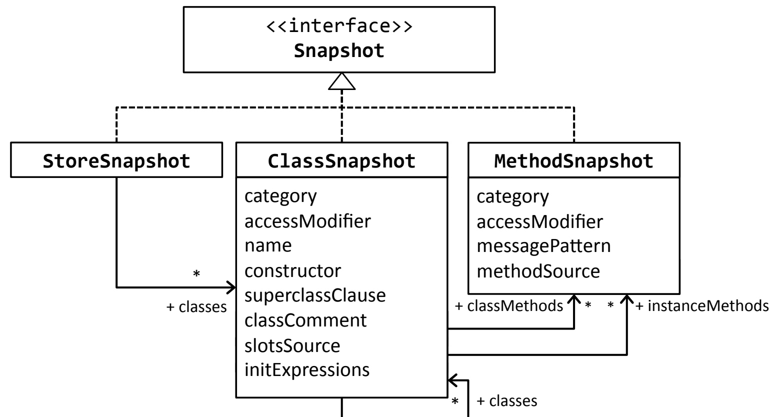

Figure 5.1: Implementation of snapshot interface for Newspeak

5.2 Stores

PNS must provide two kinds of stores, image-based stores and file-based stores. The former are required, as the Newspeak platform is implemented as an image-based IDE. In contrast to file-based systems, developers interact with live objects and classes. The state of the live classes in the image must be captured by a store. The latter is required, as the concrete implementations of Pur interact with file-based VCSs. Figure 5.2 shows the two kinds of stores.

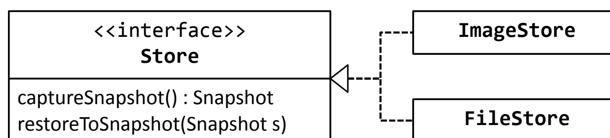

Figure 5.2: Implementations of store interface for Newspeak

5.2.1 Parsing and Printing Snapshots

Both `ImageStore` as well as `FileStore` must support reading and writing snapshots. Newspeak itself provides parsing and printing mechanisms that store live classes as files. Figure 5.3 illustrates both required and existing synchronization mechanisms. Only one of the two required synchronization mechanisms has to be implemented. The remaining mechanism can be pro-

vided by combining the other two mechanisms. PNS itself implements the `FileStore` synchronization mechanism and provides the `ImageStore` synchronization by combining the `FileStore` and file to class mechanisms provided by Newspeak.

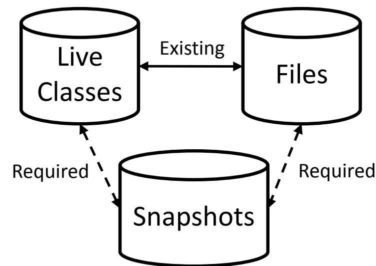

Figure 5.3: Existing and required synchronization mechanisms

Reading files into snapshots makes use of Newspeak's parser combinator framework [Bra07]. This framework allows specifying grammars separately from tools that act on them. The grammar used by the Newspeak compiler can thus also be used to build a parser that produces Snapshots. While the normal Newspeak parser produces ASTs, and thus must construct result nodes for syntactical elements as detailed as message sends, the snapshot parser produces results at the coarse-grained level of classes and methods. A custom parser for snapshots can be created by subclassing the Newspeak grammar and adding actions to create snapshots at the corresponding nodes, such as class- or method declaration. By subclassing the grammar that is also used by the AST parser, changes to the Newspeak syntax are automatically reflected in both parsers.

The snapshot parser implementation makes use of the fact that store snapshots that are to be loaded from files are usually very similar to snapshots that were loaded before. Often, only few classes change between revisions. The parser is thus wrapped in a memoizing parser that keeps a mapping of class- and method sources to snapshots. Thus, when the source of a class is parsed that has been parsed before, the snapshot can be returned with a simple lookup.

Writing snapshots back to files makes use of a custom implementation in each snapshot class that writes a standardized representation of itself to a stream or a file. This implementation strategy has the drawback that the syntactical specification of Newspeak is replicated within snapshots. Should changes to the Newspeak syntax be required, they thus must be

made in multiple places. The implementation was chosen nevertheless in order to keep the initial implementation effort low.

The `ImageStore` synchronization mechanism can be implemented by combining the existing class-to-file and file-to-snapshot mechanisms, and vice versa. This implementations strategy does not require any additional implementation to convert between snapshots and classes and can thus be realized with minimal effort. In order to speed up the conversion of live classes to snapshots, the image's change notification mechanism is employed. This mechanism allows client code to be notified when classes change. PNS keeps a mapping of live classes to snapshots, which it removes classes from when they are changed. When snapshots for all classes in the image are requested, only those for classes that have changed since the last request have to be rebuilt.

5.2.2 Stores as Working Copies

Stores form the basis of working copies. PNS treats the entire Newspeak image as a single `ImageStore` and uses this store as the current state of each working copy. For each Pur repository, PNS additionally maintains a current historian. PNS expects the classes in the image to be based on this current historian.

5.3 Diffing Algorithm

PNS must be able to show differences between snapshots. For example, in order to show the changes that were made in the Newspeak image, to browse the history of changes, or to merge versions. Several techniques to diff and merge source code exist. An overview can be found in [Men02].

PNS implements a diffing algorithm that is based on Eclipse's hierarchical diffing algorithm [CO]. Figure 5.4 shows the classes and interfaces provided by the algorithm. The algorithm diffs tree structures that consists of nodes that implement the Diffable interface. In order to be Diffable, a node must provide the following operations: It must provide access to its child nodes. It must be able to determine whether it corresponds to another node and whether it equals another node.

The algorithm compares two nodes and optionally takes a common ancestor node. It produces a DiffResult. A DiffResult consists of the two

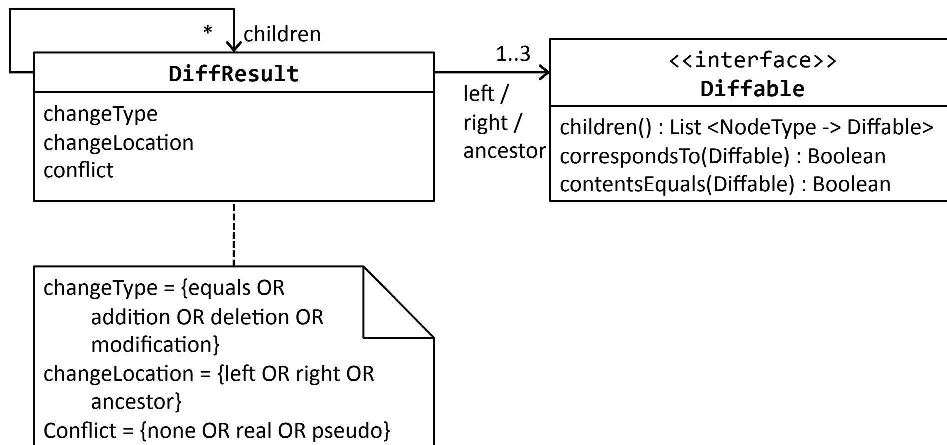

Figure 5.4: Classes of the diffing algorithm

or three input Diffable nodes, a ChangeInfo structure, as well as a set of children DiffResults. The children DiffResults are created by recursively applying the diffing algorithm to corresponding child nodes of the input nodes.

The algorithm relies on being able to identify corresponding nodes. In our implementation, nodes do correspond to each other if their parent nodes report them as belonging to the same type of nodes (e. g., instance methods or class methods) and they have the same name. As a result, the algorithm detects renamed nodes as a removal and an addition of a node. The algorithm can furthermore not detect that nodes of one type were replace by nodes of another type. For example, replacing a class by a method will be recognized as a removal and an addition.

5.4 User Interface

The user interface of PNS consists of various components that are arranged as a single coherent application to the user. Newspeak provides the application framework Hopscotch [Byk08] that allows developers to build and arrange composable user interface elements into applications which resemble web documents. Figure 5.5 shows PNS opened on a single local repository. The main user interface consists of four components. The component "Modified in Image" exposes interactions between the image and the currently loaded historian. The component "Current Local Historian"

shows the current historian in relation to other local and remote historians. The component "Other Local Historians" shows collapsed variants of the same user interface for all other local historians. The component "Remote Repositories" allows interacting with tracked remote repositories. The following sections give detailed descriptions of the single components.

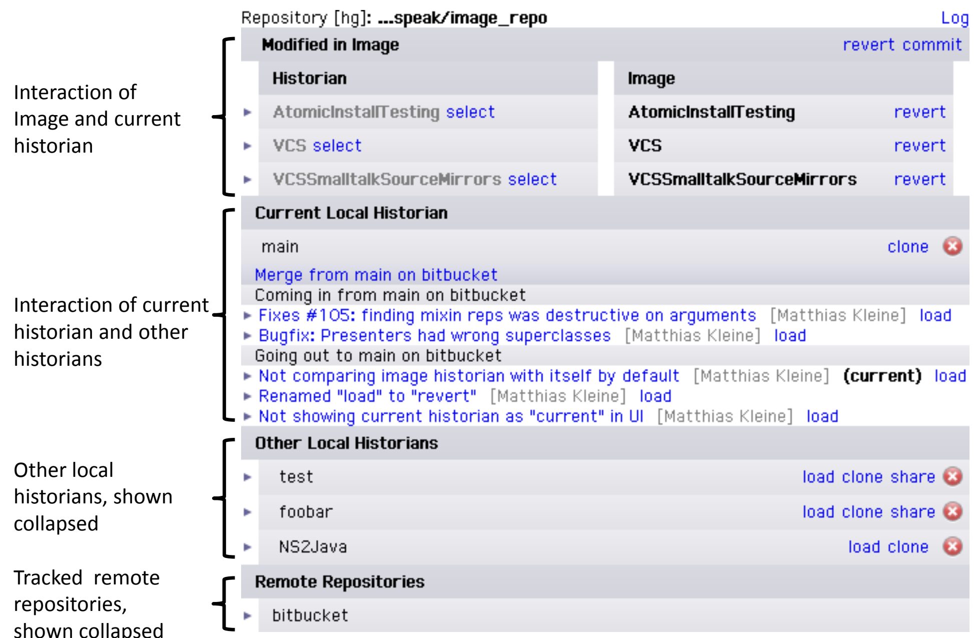

Figure 5.5: Screenshot of PNS

5.4.1 Modified in Image

The component shown in fig. 5.6 allows image changes to be analyzed, committed, or reverted. The component displays snapshots of the image store next to corresponding snapshots of the current historian, filtering out unchanged snapshots. In this example, only three classes were changed. By default differences are shown in a collapsed form, but can be expanded to expose details. Changes can be reverted to the historian's version using the "revert" links.

A commit can be initiated using the "commit" link. This link opens a text field to enter a commit message, as shown in fig. 5.7. The snapshot created by a commit can be composed of the image's and current historian's

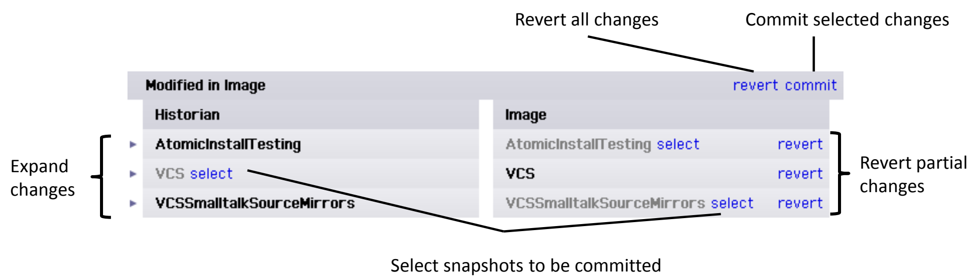

Figure 5.6: Interaction choices between image and current historian

snapshots. By default, the image store's snapshot, and thus all changes, are committed.

It can be desirable to commit only individual changes without reverting uncommitted ones. Such partial commits require developers to compose a store snapshot that includes only some of the image's changes. Developers can compose a store snapshot by choosing individual snapshots from the image's and historian's snapshot. Choosing snapshots is performed using the "select" links provided by the user interface. The example in fig. 5.6 is set up to commit only the changes made in the class "VCS", as indicated by the bold typeface and select links. The other two changed classes have the historian's snapshot selected. After committing the selected changes, the unselected changes will remain in the image and can thus be committed or reverted at a later point in time.

Figure 5.7: Changes were expanded and a commit message was entered

5.4.2 Current Local Historian

The component shown in fig. 5.8 provides the user interface for interacting with the current historian. The interface is split into a generic section and a section that compares the historian to its upstream historian. The generic section allows deleting the historian as well as creating a new historian by cloning the current one. If the historian does not have an upstream historian, the generic section also shows a link to share the historian to another repository, as illustrated in later examples.

Figure 5.8: Interaction with the current historian and related historians

The historian relation section compares the historian to its upstream historian. It shows incoming and outgoing versions and offers an appropriate action. Versions are incoming if they only exist in the upstream historian's history. Versions are outgoing if they only exist in the current historian's history. If neither incoming nor outgoing versions exist, both historians have the same version and the user is notified that the historians are in sync. If either only incoming or outgoing versions exist, the two historians are in an ancestor relation and the user interface offers an action to reset the older of the two historians to the younger historian's version. If incoming as well as outgoing versions exist the historians have diverged and the user is offered the option to merge the changes back together.

In the example shown in fig. 5.8, the upstream historian of the local historian "main" is the historian "main" on the repository "bitbucket". Two incoming and three outgoing versions exist. The developer can thus choose to merge the two historians. Doing so will bring up the merge user interface that is explained in section 5.4.5.

5.4.3 Other Local Historians

The component depicted in fig. 5.9 provides user interfaces for all other local historians. These user interfaces are extensions of the interface displayed for the current historian and are by default shown in collapsed form. In addition to the actions provided by the user interface of the current historian, the generic interface section is extended by a "load" link that makes the historian the current one and loads its version into the image. The example shown in the screen shot includes two historians that do not have an upstream historian and thus display the "share" link. When expanded, the interface shows the relation to the particular upstream historian, as well as to the historian currently loaded into the image.

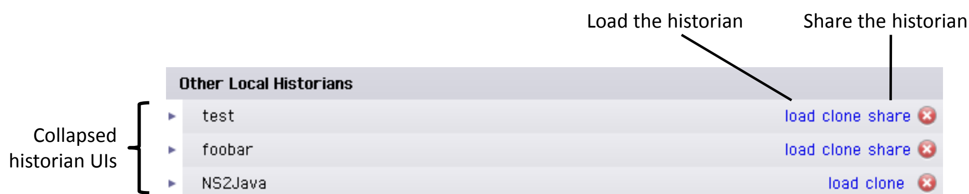

Figure 5.9: Interaction with other local historians

5.4.4 Remote Repositories

The component depicted in fig. 5.10 allows interacting with tracked remote repositories. Each tracked repository lists its historians. Historians can be "tracked", meaning they are cloned and the cloned historian's upstream historian is set to them. Historians that are already tracked are marked as such.

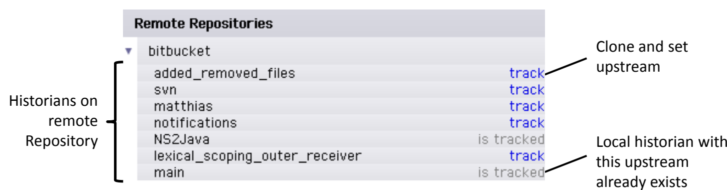

Figure 5.10: Interaction with remote repositories

5.4.5 Diffing and Merging UI

Figure 5.11 shows an excerpt of the merge UI. The interface is identical with the one used to show differences between the image and the current historian. When used in merge mode, the two versions to be merged are compared against a common ancestor. The merge UI automatically selects individual snapshots that were changed. In the screen shot, a superclass clause was changed on the left side and several snapshots were changed on the right side. When snapshots were changed in both versions, an additional column is shown that contains the ancestor's source. The developer can select any of the three columns or enter a manual merge resolution. The manual merge can be based on an automatic word-based merge.

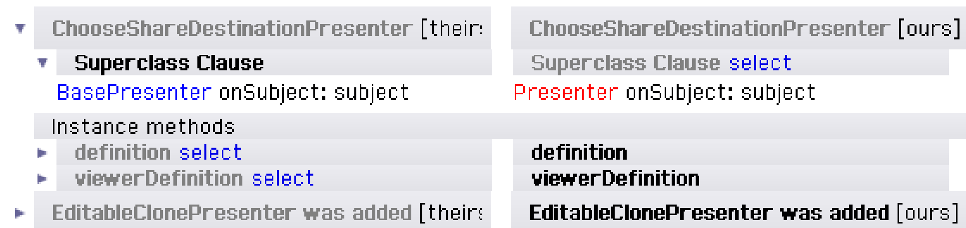

Figure 5.11: Excerpt of the merge UI

5.5 Outlook

This section identifies work flows that are common across VCSs that PNS does not support, and gives an outlook on adding support for them. Thereby, we show that the lack of supported work flows is not a result of a conceptual weakness of Pur, but instead a consequence of the prototypical implementation of PNS.

5.5.1 Projects

PNS treats the complete Newspeak image as a single store. It is desirable to introduce an abstraction for projects that allows dividing the image into multiple stores. The Newspeak image does by default have exactly one namespace for classes. With only one namespace, different projects cannot create classes with identical names. While an implementation of project-

based stores does not pose an architectural challenge, it has been deferred so that it can be implemented in accordance with an implementation of UI support for multiple namespaces.

5.5.2 History Rewriting

Git by default provides several tools that allow rewriting history by creating new versions based on existing ones. These tools include a command to amend the previously committed version. Amending a version results in a new version being created that has the amended version's parents as its own parents and thus acts as a replacement for it.

The so called rebase tool allows creating a series of versions based on an existing series of versions. It is called rebase because it is commonly used to apply a series of changes that were committed locally against new versions that were created remotely, thus changing the ancestor versions they are based against.

All of these operations result in versions being removed from the ancestry of the historian or branch they are being performed on. They are therefore often said to be "history rewriting". History rewriting operations become problematic once they affect versions that have already been shared with other people or repositories. Their usage is furthermore criticized as it results in versions being created that have not been loaded into a working copy and thus have not undergone manual testing.

PNS does not provide any support for history rewriting operations. This is not the result of Pur lacking concepts but merely of PNS lacking the appropriate user interfaces. An extension of PNS to support such operations without modifying Pur is thus possible.

5.5.3 Diff and Merge UI

PNS currently provides a hard-wired diffing algorithm that yields less than optimal results. Most VCSs can be configured to use external diffing tools. This is not possible with PNS right now. Possible improvements include allowing arbitrary diffing algorithms to be integrated or extending Pur's diffing algorithm to support renaming or refactoring detection [DMJN07, DMJN08, KN09], possibly making use of refactorings that where recorded by the IDE [HD05, DNMJ06, Fre06].

5.5.4 Bisect

Git and Mercurial provide commands that help identifying versions that introduced a bug by performing a binary search on revisions, relying on user feedback to decide whether a version contains a bug. While this is not supported by PNS, it can be implemented on top of Pur.

5.5.5 Conclusion

As seen in this section, support for various common VCS can be added to PNS without requiring modifications of Pur. The limited functionality of PNS is thus not a consequence of failures of Pur, but instead a sign of PNS's prototypical implementation status.

5.6 Summary

This chapter showed that version control applications can be built on top of Pur. PNS exposes a rich set of concepts, including branching, without relying on a specific VCSs for storing the versioned objects. As seen in this section, PNS is currently a prototypical implementation, but can be extended to support additional work flows without changing Pur.

6 Evaluation

Section 2.3 identified requirements that Pur must satisfy in order to provide an abstraction that is sufficiently rich to be the basis for client programs that perform consistently across VCS. This section evaluates to which degree Pur addresses these requirements.

6.1 Provide Rich Semantics

As motivated in section 2.3.1, Pur should provide semantics sufficiently rich to be the basis for version control client programs that perform consistently across VCS back-ends. Unlike other abstractions, Pur addresses this requirement by providing history and branching models with specified semantics. Pur history can thus be represented in any supported VCS. A detailed analysis of both history and branching models is found later in this section.

6.2 Version Control System-agnostic Interface

As motivated in section 2.3.2, Pur should not expose details of supported VCSs in order to provide uniform access to all supported VCSs. This allows building clients that work consistently across VCSs. For example, Pur clients are not required to know details of Mercurial's bookmarks or Git's branches, but can instead rely on historians.

Providing uniform access to multiple VCSs requires VCS specifics to be hidden. For example, Pur does not expose Subversion's mergeinfo and as such cannot capture the history of cherry-picking. Supporting mergeinfo would result in other undesirable consequences, see section 6.4. Given the requirement to access multiple VCSs through one abstraction, this lack of supported concepts is warranted.

6.3 Minimal Interface

As motivated in section 2.3.3, Pur should provide a minimal interface that does not specify concepts that are relevant only to specific VCSs or client programs. This is desirable as it allows establishing a set of essential version control concepts and thus prevents Pur from gaining superfluous complexity.

As a result of this conservative strategy, some concepts that are relevant to client programs were left out of Pur, such as tracked repositories or upstream historians. It is at this point not possible to evaluate whether these concepts should become part of Pur. For example, other ways for specifying relations between historians may be more desirable than upstream historians. Concrete client programs could implement custom variations.

As a further result of this strategy, concepts that are relevant only to VCSs were left out of Pur, such as version identifiers or the distinction of local and remote repositories. As a consequence, Pur implementations must address these concepts. This can cause duplicated effort. As shown in chapter 4, this does not hold for our implementation. Instead, an abstract implementation of Pur was provided that adds these missing concepts.

By being small rather than complete, Pur defers several decisions to the actual implementation. While this does have negative consequences, such as the repeated implementation effort, it is also desirable, as it prevents Pur from prematurely gaining undesirable complexity.

6.4 State-based Non-linear History Model

As motivated in section 2.3.4, Pur should provide a state-based non-linear history model in order to provide a uniform abstraction over the history models of Git, Mercurial, and Subversion. Using this history model, Pur can represent the same history in all three systems. The converse does not hold. The history models of the concrete VCSs can represent history that cannot be represented in Pur.

Pur does not have a concept of files and as such cannot capture file renamings. Unlike Pur and Git, Mercurial and Subversion can represent file re-namings. Support for representing file re-namings could be simulated in Git, by storing them as additional meta data. Support could subsequently be included in Pur. Nevertheless, the resulting benefit is not obvious. Git's

approach of determining file re-namings via heuristics delivers promising results and furthermore serves as a basis for sub-file movement detection. The increase in Pur's complexity required by this extensions is thus not warranted.

As shown in section 2.2.3, Subversion's mergeinfo attributes can represent the fact that single revisions were merged into a branch, so called cherry picking. While it is desirable to represent cherry-picking in the history, Subversion's mergeinfo-based branching has other undesirable consequences, such as the lack of support for repeated bi-directional merges. For example, if branch "branches/test" is created as a copy from "trunk", changes on "trunk" can be repeatedly merged into "branches/test", but changes on "branches/test" can only once be merged back into "trunk". Mergeinfo attributes are therefore not exposed by Pur.

6.5 Consistent Branching Model

As motivated in section 2.3.5, Pur should provide a consistent branching model in order to allow branches to function consistently across VCSs. With Historians, Pur provides a label-based branching construct that allows applying the same branching work flows across Git, Mercurial, and Subversion. Pur clients do not need to know the branching concepts of concrete VCSs.

Historians differ from the native branching models of the supported VCSs. Users of Mercurial who do not use bookmarks are used to storing branching information in changesets, respectively versions. Users of Subversion are used to manage branches manually through directories. As a result of that, Pur's work flows differ from those of the concrete VCSs. Yet, losing VCS-specific branching concepts gains the benefit of providing a single branching construct with rich semantics that can be used consistently across VCSs.

6.6 Conclusion

Pur successfully addresses all the requirements set up in section 2.3. As such, it provides an abstraction that is sufficiently rich to be the basis for client programs that perform consistently across VCS. As seen in this evaluation, providing a uniform abstraction results in a certain loss of

control over the details of supported VCSs. Consequently, Pur is not an appropriate abstraction for use-cases that rely on VCS-specific concepts or work flows. In contrast, use cases that explicitly desire to apply the same concepts and work flows across various VCSs find Pur to be a valuable solution.

7 Related Work

Abstractions over version control have been built in various contexts. The wider context of Software Configuration Management (SCM) has seen research on establishing general principles of change in software, but not with the intent of building an interface for multiple existing VCSs. Some research in the field of version control exists that aims to provide an abstraction for version control concepts. Finally, several implementations exist that aim to address this goal to some extent. This chapter analyzes both the work established in the scientific community as well as concrete implementation and relates both of them to Pur.

7.1 Software Configuration Management

Several attempts to capture general principles of change in software exist within the wider research field of Software Configuration Management (SCM). SCM is concerned with the control of the evolution of complex systems [Est00], which version control is an important part of [MWE10].

In [CW98], a taxonomy of fundamental version control concepts is established and existing SCM systems are categorized using this taxonomy. The taxonomy is not built with the intent of establishing an implementable interface, but with the goal of identifying methods to describe configurations of versioned artifacts through rules, so called intensional versioning. As such, a large and diverse number of SCM systems are compared. The resulting taxonomy covers a broad range of version control concepts. It identifies representations and relations of concepts both for objects being versioned, the so called product space, and objects that represent the versioning, the so called version space. Pur addresses only a fraction of these concepts, but does so with a different intent.

A taxonomy for change in software is presented in [BMZ+05]. This taxonomy is intended to cover a broad range of aspects, including aspects that are related to version control, such as change history, as well as aspects that are often not found in VCSs, such as type of change. Unlike Pur, the

goal of this taxonomy is not to be the basis for a concrete software interface for version control but instead to allow comparisons between tools that support change in software, such as refactoring tools as well as VCSs to be made.

[WMC01] aims to provide a unified model for version management, not with the attempt to support several existing version control systems, but with the intent of providing a core implementation that can be extended by custom implementations of history models or version storage. By not relying on a state-based history model, this model allows expressing more general VCSs. It additionally adds support for configuration rules that are commonly found in SCM research.

In [Zel97], a formal model of SCM is presented with the goal of establishing a SCM system that can be adapted to the user's needs. The model uses a single logical formalism to describe various SCM concepts, such as revisions, variants, workspaces, and configurations. Using this unified model, an experimental SCM system is built that is used to show the configurability of the approach.

In summary, existing research on SCM abstractions does not aim to provide an implementable abstraction that allows client programs to interact with multiple systems. Instead, existing research is concerned with establishing general models that can be used to either compare existing systems or build new systems.

7.2 Version Control

Some attempts to formalize version control outside the scope of SCM exist. The authors of [LSL] provide a formal specification of version control that is based on the concept of patches. Patches are rules that specify how a repository is modified. A formal theory is presented that describes when patches can be applied. It lacks essential version control concepts such as branches. This theory is kept abstract and not intended to be mapped to existing VCSs.

In [Bra09], initial ideas for an abstraction over version control systems that is to serve as an interface for VCS-agnostic clients is presented. As such, the goals of this paper are very similar to that of this report. Some concepts of the resulting abstraction have found its way into Pur, such as the concept of a store. The abstraction handles other concepts differently

than Pur. Versions are separated from their history. A distinction between servers and clients is made, where clients store only a linearized history. These differences are analyzed in the next paragraphs.

Pur has the concept of versions that have parents whereas the related model has two separate concepts of version and history. In the related model a version does not know its parents. Pur unifies the concepts of version and history for the following reasons: The meta data that is part of a version describes how the version was created in relation to previous versions. For example, a commit message describes the changes that were made to previous versions. The author indicates who committed these changes. None of the meta data that is attached to a version makes sense without the history information. The history of a version is a integral part of a version.

The related model introduces two kinds of peers, clients and servers, with clients being limited to storing linear history and servers storing history as a DAG, but being limited to having one childless version. Pur has only one type of peer, repositories that also can have multiple childless versions but introduces historians to organize access to branches.

The distinction of servers and clients was made in the related abstraction with the intent of supporting clients with limited storage capacities. Pur evades this problem by leaving unspecified where versions are stored. Pur requires that versions can be accessed via historians through repositories, but does not require repositories to provide their own version store. Instead it is also possible to implement a Pur system that allows repositories to be owned by clients with limited storage capabilities by storing only some versions on the actual client and deferring access to unavailable versions to remote repositories.

In conclusion, only few formalized and implementable abstractions over VCSs exist. Most abstractions over VCSs are not designed as implementable interfaces and address other goals, such as identifying ways to specify configurations. No formalized and implementable abstraction is known to us that has actually been implemented. Pur thus contributes by specifying an abstraction that is shown to be implementable.

7.3 Implementations of Version Control System Abstractions

Tools that can interact with multiple VCSs exist. Each of these tools provides some form of abstraction over VCSs. Across these implementations, different originating requirements can be identified. For example, some abstractions expose rich interfaces and include concepts of history and branching; others are shallow and require implementors to expose VCS-details. This section analyzes the concrete implementations by identifying the requirements that they address, evaluating the implementation, and relating them to Pur.

7.3.1 Eclipse

Eclipse is an IDE that allows developing software in several programming languages. Eclipse's Team component[1] allows interacting with various VCSs. Implementations exist for CVS, Subversion, Git, Mercurial, and possibly others.

The goal of the Team component is to allow building custom version control applications that are integrated into the IDE. Part of this goal is to avoid enforcing a work flow on resulting applications. The focus correspondingly lies on providing means to interact with the IDE, e. g. allowing to register callbacks for file change notifications or accessing the contents of files at other revisions. The Team component provides a simple history model, but no abstraction for branches.

The Team component's history model can capture non-linear history on a file-basis. This differs from Pur, which can capture history on a store-basis. Eclipse separates history into revisions and their history. The interface IFileRevision provides access to a file's revision. A revision consists of a snapshot of the file and additional meta data, such commit message, author, and time stamp. Revisions are separated from history. The interface IFileHistory provides access to the history graph linking revisions. It exposes a revision's child and parent revisions. A revision can have multiple parent revisions, so called contributors, as well as child revisions, so called targets.

[1] `http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/team.htm` – last checked 20.10.2010

These interfaces can be implemented to some degree by back-ends that capture history on a project-basis. They must be able to trace changes to single files across revisions. Subversion and Mercurial internally store changes on a file basis and thus can track file renames. Unlike that, systems like Git have to rely on heuristics.

Version control applications built on the basis of the Eclipse Team abstractions can make use of tools provided by Eclipse, such as a hierarchical diff viewer (see fig. 7.1). Making use of the diff viewer relies on exposing the contents of a file at a revision in a format usable by the diff viewer.

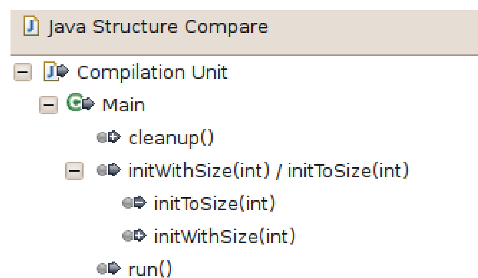

Figure 7.1: Eclipse's hierarchical diffing UI

Version control applications can be fitted to custom VCS concepts and work flows. Thus, a Git back-end can provide tools to interact with Git's branches and can furthermore adopt Git's selective committing approach, whereas a Mercurial back-ends can adopt appropriately.

In comparison to Pur, the abstraction provided by Eclipse's Team component lacks concepts for branching development as well as for history on a store basis. This results in fewer shared user interfaces and thus in a higher amount of duplicated effort.

7.3.2 IntelliJ IDEA

IntelliJ IDEA[2] is a commercial IDE developed by JetBrains. It provides an abstraction for VCSs that is implemented for Subversion, Git, and Mercurial. As the Eclipse Team abstraction, this abstraction is designed as a basis of custom version control applications and as such provides a set of tools, such as history or diff viewers, that can be used across VCS integrations.

[2] http://www.jetbrains.com/idea/ – last checked 26.10.2010

IntelliJ IDEA provides a history model that separates history from revisions. A VcsFileRevision provides access to the contents of a file at a certain revision, together with meta data such as author, time stamp, and commit message. A VcsHistoryProvider can open a VcsHistorySession that allows listing past revisions of a file. The history model is at this point linear. The generic history viewer thus can only show a list of revisions. No abstraction for branches exists. As such, VCS back-ends implement their own abstractions and user interfaces.

Figure 7.2 shows the menus that the integrations for Subversion and Git provide. As can be seen, the interaction possibilities are customized to the back-ends. For example, the Git integration allows users to make use of Git's rebase operation, which rewrites history. These operations are not available in the Subversion back-end.

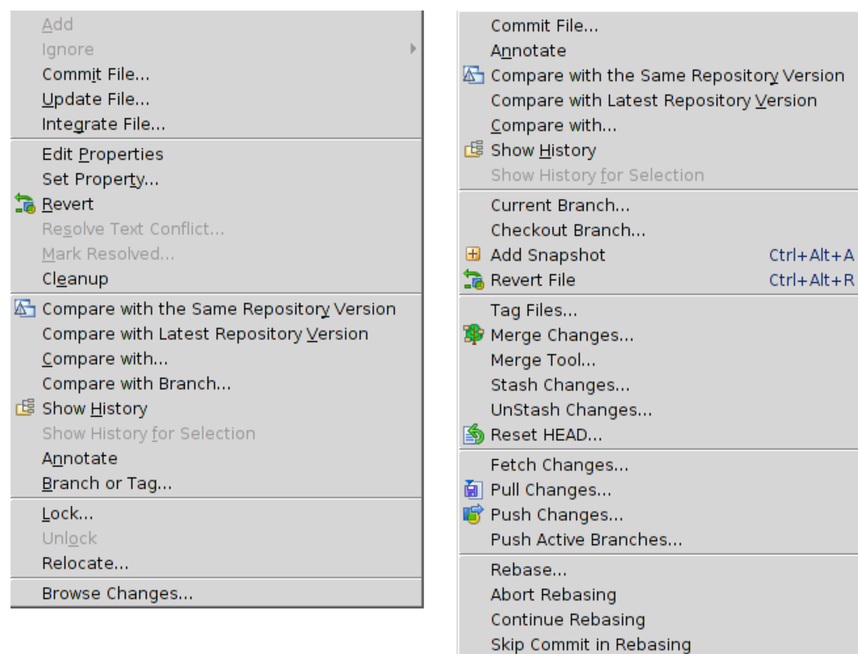

Figure 7.2: IntelliJ IDEA's VCS-interaction menus for Subversion (left) and Git (right)

In conclusion, IntelliJ IDEA allows building VCS-integrations that provide tools custom to each back-end. Interaction with common tools is provided in a limited fashion. The lack of abstractions for non-linear history or branches hinders the implementation of history viewers that would be appropriate for Mercurial and Git.

7.3.3 Emacs DVC

GNU Emacs [Sta02] is a text editor that is designed to be extended by its users. Several extensions to interact with VCSs exist. This section analyzes DVC[3], a version control application that acts as a front-end for various VCSs by providing a single set of user-interface operations. DVC is implemented for several back-ends, including GNU Arch, Bazaar, Git, Mercurial, and Monotone.

DVC relies on a common abstraction that must be implemented by every back-end that is very close to the actual user interface provided by DVC. As such, it includes operations to print a revision, to print the diff between revisions, or to switch to another branch.

DVC does not require back-ends to provide structured access to revisions besides being able to list and print them. This allows DVC to support VCSs with different history models. For example, implementations for systems that are not based on snapshots but on patches, such as Darcs [Rou05], can be implemented. This is different from Pur, which assumes that history can be represented as a directed acyclic graph.

DVC stands out of the other VCS abstractions by providing a common interface to interact with branches. Back-ends must implement operations to create a new branch and to switch branches. This interface carries only limited semantics about what a branch actually is and thus is not restricted to any branching model. For example, the Git back-end implements branches using Git's label-based branches whereas the Mercurial back-end uses Mercurial's commit-message based branches. As a consequence, a certain knowledge of the back-end is required to know how branches behave.

When compared to Pur, DVC can support a wider variety of VCSs. This is achieved through an abstraction that leaves details about history- and branching model unspecified. This at the same time limits the applicability of DVC. Its abstractions are designed to be the base of one concrete version control application. The lack of a clear separation of an intermediate model and the version control application itself hinders the implementation of other applications on top of DVC. Finally, the semantics of branches depend on the back-end and thus require the user to be familiar with it.

[3] http://download.gna.org/dvc/ – last checked 20.10.2010

7.3.4 Other Abstractions

Various other VCS abstractions exist that to some extent aim to address the requirements addressed by Pur. The following list of implementations of VCS abstractions is presented in less detail than the ones presented so far, either because they aim to address other goals than Pur or because their applicability is limited.

anyvc Anyvc[4] is a library that provides an abstraction to interact with various VCSs. It has been implemented for Bazaar, Mercurial, Git, and Subversion. Anyvc is being used by the Pida IDE[5].

Anyvc provides an abstraction for non-linear history similar to Pur's. A Revision has parents, meta information, and allows accessing files. An abstraction for branches does not exist yet, but is considered future work. As such, Anyvc does not address the requirement of providing rich semantics.

pyvcs Pyvcs aims to be a "minimal VCS abstraction layer for Python"[6]. Its goal is to provide an abstraction to allow browsing code at arbitrary revisions in various VCSs. As such, it does not implement abstractions for branches. It is implemented for Mercurial, Git, Subversion, and Bazaar.

MR MR [7] is a "Multiple Repository management tool". It allows running identical commands across multiple repositories. For example, it allows committing changes made in several working copies at once. MR can run commands across repositories of different VCS, providing support for Subversion, Git, CVS, Mercurial, Bazaar, and Darcs.

The commands exposed by MR are mapped to commands of the concrete VCSs. MR supports commands, such as "update", "status", "commit", and "diff", but does not include any branching abstractions. As such, it does not address the requirement for providing rich semantics.

VCI—The Version Control Interface VCI[8] is a library that provides an abstraction for various VCSs. It has been implemented for Bazaar, Mercurial, Git, CVS, and Subversion. VCI provides an abstraction for history, but none for branches.

---

[4] http://bitbucket.org/RonnyPfannschmidt/anyvc/ – last checked 25.10.2010
[5] http://pida.co.uk/ – last checked 25.10.2010
[6] http://github.com/alex/pyvcs/blob/master/README.txt – last checked 25.10.2010
[7] http://kitenet.net/~joey/code/mr/ – last checked 15.11.2010
[8] http://vci.everythingsolved.com – last checked 25.10.2010

It separates abstractions for revisions and history. Revisions can be accessed using a structured interface that exposes information such as committed data, author, and time stamp. The history abstraction supports only linear history. According to VCI's documentation, abstractions for both branching and non-linear history are considered future work.

Visual Studio Microsoft Visual Studio[9] is an IDE developed by Microsoft. It provides two ways to integrate VCSs into it[10]. First, by implementing a custom user interface that does not rely on any shared version control abstractions by creating a so called VSPackage. Lacking any abstractions, this approach allows complete control over the integration but at the same time does have none of the benefits of a shared abstraction.

The second approach to integrate a VCS is to implement a so called Source Control Plug-in. Visual Studio provides a minimal source control user interface that allows initiating interaction with a plug-in. For example, a user can request to see the history of a file. A source control plug-in must provide call backs that are invoked when a user requests one of these actions. It is the plug-ins responsibility to construct an appropriate user interface. As such, a plug-in must provide most of its own user interface. The set of actions is fixed and contains actions that are not relevant for all VCSs, for example actions to lock files exist, but no actions to synchronize repositories exist. As such, this approach is limited in the set of VCSs that it supports.

7.3.5 Summary

Several implementations of version control abstractions exist. None of these abstractions addresses all of Pur's requirements. Most abstractions rely on implementations exposing VCS-specific concepts. Only few of these abstractions provide constructs for both history and branching. No abstractions exists that provides a branching model with semantics specified to the degree that Pur does.

[9] `https://www.microsoft.com/visualstudio/en-us/` – last checked 26.10.2010
[10] `http://msdn.microsoft.com/en-us/library/bb164701(v=VS.80).aspx` – last checked 26.10.2010

8 Summary and Outlook

This report presents the version control abstraction Pur and evaluates it for practicability and applicability. This chapter gives a summary of the contributions presented by this report and based on that gives an outlook on future work.

Pur captures essential version control concepts as a set of interfaces that can be implemented for concrete VCSs, such as Git, Mercurial, and Subversion. Pur allows version control clients, such as IDE tools or web status reports, to support all of these concrete VCSs through a single set of interfaces and thus eliminates the need to build custom implementations for each VCS. Unlike most other abstraction for VCSs, Pur exposes constructs that help organizing branching development.

Pur has been tested for practicability by implementing it within the Newspeak programming platform. Implementations exist for Git and Mercurial, with a implementation strategy for Subversion having been described. An evaluation of these implementations for applicability was performed by implementing the version control application PNS as a Pur client.

The research on Pur has led to new questions becoming relevant. The following paragraphs give an outlook on possible next steps.

Implement Support for Subversion This report proposes a strategy for implementing Pur for Subversion. This strategy has not been tested for practicability. As of such, final judgment must be deferred until concrete implementations exist.

Build Other Pur Clients The version control application PNS is currently the only client that makes use of Pur. Pur is designed to provide an appropriate abstraction for various kinds of VCS clients. Possible other clients include web-based status report tools or information extraction tools. While these tools are expected to be implementable on top of Pur, a concrete evaluation can only performed once such implementations exist.

Extend Pur Pur is not intended to solve all version control abstraction challenges. Yet, Pur can serve as a basis for experimentation with solutions for remaining issues, such as finding abstractions that are relevant to only some VCSs or client programs. This report has already shown first ideas for such extensions. Our implementation extends Pur both on the back-end side, through the distinction of local and remote repositories, as well as on the client program side, through extensions such as upstream historians. All of these extensions require further evaluation. Additional Pur implementations and client programs are needed in order to better evaluate the applicability of such extensions.

Bibliography

[BAB+08] G. Bracha, P. Ahe, V. Bykov, Y. Kashai, and E. Miranda. The Newspeak Programming Platform. Technical report, Cadence Design Systems, 2008.

[Bla06] J. Black. Compare-by-Hash: A Reasoned Analysis. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, pages 85–90, Berkeley, CA, USA, 2006. USENIX Association.

[BMZ+05] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a Taxonomy of Software Change. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5):309–332, 2005.

[Bra07] Gilad Bracha. Executable Grammars in Newspeak. *Electronic Notes in Theoretical Computer Science*, 193:3–18, 2007.

[Bra09] Gilad Bracha. Source Control and Synchronization. Unpublished Manuscript, February 2009.

[Byk08] Vassili Bykov. Hopscotch: Towards User Interface Composition. In *1st International Workshop on Academic Software Development Tools and Techniques*, 2008.

[Car98] Antonio Carzaniga. Design and Implementation of a Distributed Versioning System. Technical report, Dipartimento di Elettronica e Informazione, Piazza Leonardo da Vinci, 32, 20133 Milano, Italy, 1998.

[CO] Eclipse Contributors and Others. Eclipse Documentation: Class Differencer. `http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/org/eclipse/compare/structuremergeviewer/Differencer.html`.

[CW98] Reidar Conradi and Bernhard Westfechtel. Version Models for Software Configuration Management. *ACM Computing Surveys*, 30(2):232–282, June 1998.

[DAS09] Brian De Alwis and Jonathan Sillito. Why Are Software Projects Moving From Centralized to Decentralized Version Control Systems? *Cooperative and Human Aspects on Software Engineering, 2009. CHASE'09. ICSE Workshop on*, pages 36–39, 2009.

[DMJN07] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N. Nguyen. Refactoring-Aware Configuration Management for Object-Oriented Programs. In *29th International Conference on Software Engineering*, pages 427–436. IEEE, May 2007.

[DMJN08] Danny Dig, Kashif Manzoor, Ralph E. Johnson, and Tien N. Nguyen. Effective Software Merging in the Presence of Object-Oriented Refactorings. *IEEE Transactions on Software Engineering*, 34(3):321–335, 2008.

[DNMJ06] Danny Dig, Tien N. Nguyen, Kashif Manzoor, and Ralph Johnson. MolhadoRef: A Refactoring-aware Software Configuration Management Tool. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, pages 732–733, New York, NY, USA, 2006. ACM.

[Est96] Jacky Estublier. Work Space Management in Software Engineering Environments. In Ian Sommerville, editor, *Software Configuration Management*, volume 1167 of *Lecture Notes in Computer Science*, chapter 9, pages 127–138. Springer Berlin / Heidelberg, Berlin/Heidelberg, 1996.

[Est00] Jacky Estublier. Software Configuration Management: A Roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 279–289, New York, NY, USA, 2000. ACM.

[Fre06] Tammo Freese. Refactoring-aware Version Control. In *Proceedings of the 28th International Conference on Software Engineering*, pages 953–956, New York, NY, USA, 2006. ACM.

[HD05] Johannes Henkel and Amer Diwan. CatchUp!: Capturing and Replaying Refactorings to Support API Evolution. In *Proceedings of the 27th International Conference on Software Engineering*, pages 274–283, New York, NY, USA, 2005. ACM.

[KKP07] Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce. A Formal Investigation of Diff3. In *Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science*, FSTTCS'07, pages 485–496, Berlin, Heidelberg, 2007. Springer-Verlag.

[KN09] M. Kim and D. Notkin. Discovering and Representing Systematic Code Changes. In *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 309–319. IEEE Computer Society, 2009.

[LSL] Andres Löh, Wouter Swierstra, and Daan Leijen. A Principled Approach to Version Control. `http://people.cs.uu.nl/andres/VersionControl.html`.

[Mac06] Matt Mackall. Towards a better SCM: Revlog and Mercurial. In *Proceedings of the Linux Symposium (Ottawa, Ontario, Canada)*, volume 2, pages 83–90, 2006.

[Mal10] Carl F. Malmsten. Evolution of Version Control Systems. Bachelor's Thesis. *University of Gothenburg*, May 2010.

[Men02] T. Mens. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, May 2002.

[MWE10] Leonardo G. Murta, Claudia M. Werner, and Jacky Estublier. The Configuration Management Role in Collaborative Software Engineering. In Ivan Mistrík, John Grundy, André van der Hoek, and Jim Whitehead, editors, *Collaborative Software Engineering*, pages 179–194. Springer, first edition, March 2010.

[OG90] Brian O'Donovan and Jane B. Grimson. A Distributed Version Control System for Wide Area Networks. *IEEE Software Engineering Journal*, 5(5):255–262, 1990.

[O'S09] Bryan O'Sullivan. Making Sense of Revision-control Systems. *Communications of the ACM*, 52(9):56–62, 2009.

[Rou05] David Roundy. Darcs: Distributed Version Management in Haskell. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 1–4, New York, NY, USA, 2005. ACM Press.

[Sch07] Carey Schwaber. The Forrester Wave™: Software Change And Configuration Management, Q2 2007. Forrester Research Inc., May 2007.

[Sta02] Richard M. Stallman. *GNU Emacs Manual, For Version 21, 15th Edition*. Free Software Foundation, 15 edition, August 2002.

[WMC01] Bernhard Westfechtel, Bjørn Munch, and Reidar Conradi. A Layered Architecture for Uniform Version Management. In *IEEE Transactions on Software Engineering*, pages 1111–1133. IEEE Computer Society, 2001.

[Zel97] Andreas Zeller. *Configuration Management with Version Sets*. PhD thesis, Technische Universität Braunschweig, April 1997.

Colophon

This report was typeset by LaTeX 2ε with pdf/ε-TeX using KOMA-Script. The body text is set 11/14 pt on a 30 pc measure. The body type face is *Palatino* by Hermann Zapf, brought to TeX as Type 1 PostScript font *URW Palladio L*. The listing type face is *Bera Mono*, based on the *Vera* family by Bitstream, Inc.; Type 1 PostScript fonts were made available by Malte Rosenau and Ulrich Dirr.

—Tobias Pape

Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren / Redaktion
53	978-3-86956-160-8	**Web-based Development in the Lively Kernel**	Jens Lincke, Robert Hirschfeld (Eds.)
52	978-3-86956-156-1	**Einführung von IPv6 in Unternehmensnetzen: Ein Leitfaden**	Wilhelm Boeddinghaus, Christoph Meinel, Harald Sack
51	978-3-86956-148-6	**Advancing the Discovery of Unique Column Combinations**	Ziawasch Abedjan, Felix Naumann
50	978-3-86956-144-8	**Data in Business Processes**	Andreas Meyer, Sergey Smirnov, Mathias Weske
49	978-3-86956-143-1	**Adaptive Windows for Duplicate Detection**	Uwe Draisbach, Felix Naumann, Sascha Szott, Oliver Wonneberg
48	978-3-86956-134-9	**CSOM/PL: A Virtual Machine Product Line**	Michael Haupt, Stefan Marr, Robert Hirschfeld
47	978-3-86956-130-1	**State Propagation in Abstracted Business Processes**	Sergey Smirnov, Armin Zamani Farahani, Mathias Weske
46	978-3-86956-129-5	**Proceedings of the 5th Ph.D. Retreat of the HPI Research School on Service-oriented Systems Engineering**	Hrsg. von den Professoren des HPI
45	978-3-86956-128-8	**Survey on Healthcare IT systems: Standards, Regulations and Security**	Christian Neuhaus, Andreas Polze, Mohammad M. R. Chowdhuryy
44	978-3-86956-113-4	**Virtualisierung und Cloud Computing: Konzepte, Technologiestudie, Marktübersicht**	Christoph Meinel, Christian Willems, Sebastian Roschke, Maxim Schnjakin
43	978-3-86956-110-3	**SOA-Security 2010 : Symposium für Sicherheit in Service-orientierten Architekturen ; 28. / 29. Oktober 2010 am Hasso-Plattner-Institut**	Christoph Meinel, Ivonne Thomas, Robert Warschofsky et al.
42	978-3-86956-114-1	**Proceedings of the Fall 2010 Future SOC Lab Day**	Hrsg. von Christoph Meinel, Andreas Polze, Alexander Zeier et al.
41	978-3-86956-108-0	**The effect of tangible media on individuals in business process modeling: A controlled experiment**	Alexander Lübbe
40	978-3-86956-106-6	**Selected Papers of the International Workshop on Smalltalk Technologies (IWST'10)**	Hrsg. von Michael Haupt, Robert Hirschfeld
39	978-3-86956-092-2	**Dritter Deutscher IPv6 Gipfel 2010**	Hrsg. von Christoph Meinel und Harald Sack
38	978-3-86956-081-6	**Extracting Structured Information from Wikipedia Articles to Populate Infoboxes**	Dustin Lange, Christoph Böhm, Felix Naumann
37	978-3-86956-078-6	**Toward Bridging the Gap Between Formal Semantics and Implementation of Triple Graph Grammars**	Holger Giese, Stephan Hildebrandt, Leen Lambers

ISBN 978-3-86956-158-5
ISSN 1613-5652