alpha-VVS: Ein integriertes Versionsverwaltungssystem als Baustein einer Prozessunterstützung auf Basis von aktiven Dokumenten

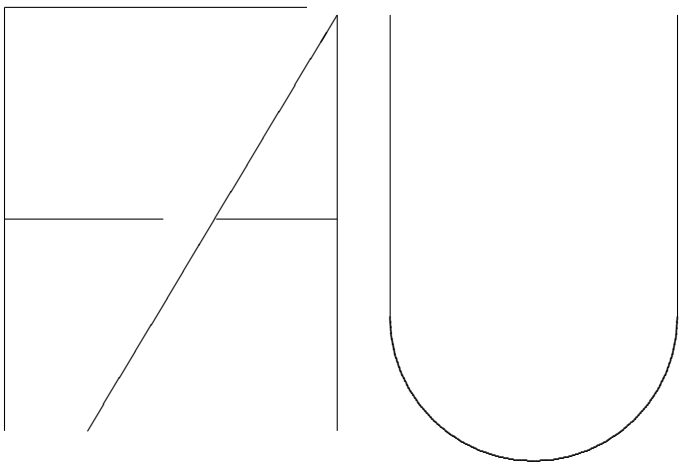
Diplomarbeit

Scott Allen Hady

Lehrstuhl für Informatik 6
(Datenmanagement)

Department Informatik
Technische Fakultät

Friedrich Alexander-
Universität
Erlangen-Nürnberg



alpha-VVS: Ein integriertes Versionsverwaltungssystem als Baustein einer Prozessunterstützung auf Basis von aktiven Dokumenten

Diplomarbeit im Fach Informatik

vorgelegt von

Scott Allen Hady

geb. 28.10.1978 in Viroqua, Wisconsin - U.S.A.

angefertigt am

Department Informatik
Lehrstuhl für Informatik 6 (Datenmanagement)
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: Univ.-Prof. Dr.-Ing. habil. Richard Lenz
Dipl.-Inf. Christoph P. Neumann

Beginn der Arbeit: 01.05.2011
Abgabe der Arbeit: 01.11.2011

Erklärung zur Selbständigkeit

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass diese Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäßübernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Informatik 6 (Datenmanagement), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Diplomarbeit einschlielich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 01.11.2011

(Scott Allen Hady)

Kurzfassung

alpha-VVS: Ein integriertes Versionsverwaltungssystem als Baustein einer Prozessunterstützung auf Basis von aktiven Dokumenten

Heutige Versionskontrollsysteme (VCSs) sind hochentwickelte Systeme, die bei der Softwareentwicklung und auch bei der Unterstützung von verteilten und simultan Bearbeitung von elektronischen Dokumenten eine entscheidende Rolle spielen. Dennoch fehlt ihnen in zwei wichtigen Bereichen support: (1) Unabhängige Entwicklung von Subprojekten und (2) die Unterscheidung zwischen Versionen, die auf deren Gültigkeitseigenschaften basiert.

Da es nicht möglich ist, auf die verschiedenen, vorangegangenen Zustände für die einzelnen arbeitenden Elemente oder Sets zurückzugreifen, wird die Autonomie jedes Einzelnen eingeschränkt. Ohne diese Autonomie hängt der Fortschritt eines jeden einzelnen Elements von dem langsamsten ab.Genauso beeinflusst das Verändern jedes einzelnen Elements alle anderen.Wenn die Status eines jedes Arbeitselements unabhängig von den anderen verwaltet werden können, wird die gegenseitige Abhängigkeit verringert und die Autonomie verbessert.

Das Arbeiten mit inkorrekten oder ungültigen Informationen ist voller Risiken und führt im Allgemeinen zu einer suboptimalen Lösung. Ein Doktor, der aufgrund von falschen Informationen eine Diagnose stellt, riskiert die Gesundheit seines Patienten. So läuft auch ein Softwareentwickler Gefahr, zu versagen, wenn er aus fehlerhaften Quellen heraus arbeitet. Deswegen ist es wichtig, gespeicherte Informationen basierend auf deren Gültigkeitseigenschaften zurück zu verfolgen und zu differenzieren.

Alpha-Flow, ein distributed Document-oriented Process Management system zur Unterstützung von inter-institutioneller medizinischer Versorgung, liefert ein konkretes Beispiel für ein System, das diese versioning capabilities benötigt. Jede Alpha-Card ist ein Set von elektronischen Dokumenten, die eine spezifische Bearbeitungsinteraktion repräsentieren und die eine unabhängige Vergangenheit beibehalten müssen, sodass seine Autonomie innerhalb des gesamten Bearbeitungskontexts gewährleistet wird. Zusätzlich senkt der Verzicht auf den Gebrauch von ungültigen Informationen die Wahrscheinlichkeit, dass ein Mediziner einen Fehler macht.

Dieses Projekt zeigt das Design und die Umsetzung einer platformunabhängigen, logischen einheitsorientierten VCS, Hydra, welches beide dieser Konzepte unterstützt. Es liefert, bei minimalem benötigten Speicherplatz, eine Funktionalität, die dafür geeignet ist, Versionskontrollsupport in andere Anwendungen einzubauen und bietet auch ein user interface, das seine Anwendung als allein-operierendes VCS unterstützt.

Abstract

alpha-VVS: An integrated Version Control System as a Component of Process Support based on Active Documents

Contemporary Version Control Systems (VCSs) are highly-evolved software systems; playing a critical role in software development and supporting distributed and concurrent collaborative effort over a set of electronic documents. However, they lack support in two important areas: (1) independent management of multiple histories and (2) differentiation between versions based on their validity characteristics.

The inability to support independent histories for multiple sets of electronic documents restricts the autonomy of each. Without its autonomy, the progress of each set is reduced to the progress of the slowest. Likewise, any change made to a by one directly affects the others. Managing the history of each working set independently reduces the interdependencies and improves autonomy.

Working from incorrect or invalid information is risky and generally leads to a suboptimal solution. A doctor making a diagnosis based on incorrect information may risk the health of the patient. Likewise, a software developer working from faulty source code is more likely to fail. Thus it is important to track and differentiate recorded information based on its validity characteristics.

Alpha-Flow, a distributed Document-oriented Process Management system that aims to support inter-institutional health care processes, provides a concrete example of a system that requires these versioning capabilities. Each alpha-Card is a set of electronic documents which represent a specific treatment interaction and must maintain an independent history to ensure its autonomy within the overall treatment episode's context. Additionally, avoiding usage of invalid information reduces the chances of a medical professional making a mistake.

This project presents the design and implementation of a platform independent Multi-Headed VCS, Hydra, which supports both of these concepts. It provides a lightweight core of functionality appropriate for embedding version control support within Alpha-Flow as well as a user interface that supports its employment as a standalone VCS.

Table of Contents

Cover 2

Title 4

Declaration 6

Kurzfassung 8

Abstract 10

Table of Contents i

List of Figures vii

List of Tables xi

List of Abbreviations xiii

1 Introduction 1
- 1.1 Project Motivation . . . 2
  - 1.1.1 Alpha-Flow Project Overview . . . 2
  - 1.1.2 Alpha-Flow's Key Characteristics . . . 4
  - 1.1.3 Multi-Headed Versioning . . . 4
  - 1.1.4 Version Validity . . . 6
- 1.2 Project Goal . . . 6

2 Method 7

3 Requirements 9
- 3.1 Requirements Management . . . 9
- 3.2 Functional Requirements . . . 9
  - 3.2.1 Basic Version Control Capabilities . . . 9
  - 3.2.2 All Versions of Artifacts Must be Maintained Locally . . . 10
  - 3.2.3 Alpha-Doc and Alpha-Card Versioning . . . 10
  - 3.2.4 Version Differentiation Based on Validity Characteristics . . . 10
  - 3.2.5 Support for Alpha-Flow Global Conflict Resolution Schemes through History Manipulation . . . 10
  - 3.2.6 Produce a Visual Depiction of Histories . . . 11

3.3 Non-Functional Requirements and Constraints . . . 11
3.3.1 No Human Interaction . . . 11
3.3.2 Platform Independence . . . 11
3.3.3 Java-Based Implementation . . . 12
3.3.4 Maven Project Integration . . . 12
3.3.5 Lightweight . . . 12

4 Analysis – Alpha-Flow 13
4.1 Fundamental Concepts . . . 13
4.1.1 Case Handling Paradigm . . . 13
4.1.2 Document-Centered Collaboration . . . 14
4.2 Domain Model . . . 15
4.2.1 Alpha-Docs . . . 15
4.2.2 Alpha-Cards . . . 15
4.2.3 Alpha-Adornments . . . 16
4.2.4 Domain Model Summary . . . 17
4.3 System Architecture . . . 18
4.4 Off-Line Synchronization . . . 20
4.4.1 Detecting Synchronization Anomalies . . . 20
4.4.2 Reconciling Detected Anomalies . . . 21
4.5 Alpha-VerVarStore . . . 23
4.5.1 Current Implementation . . . 23
4.5.2 Shortcomings . . . 23
4.6 Summary . . . 24

5 Analysis – Version Control Systems and Their Evolution 25
5.1 Source Code Control System . . . 25
5.1.1 Deltas . . . 26
5.1.2 Other Important Concepts . . . 27
5.2 Revision Control System . . . 28
5.2.1 Reverse Deltas . . . 28
5.2.2 Branching . . . 29
5.2.3 Other Important Concepts . . . 29
5.3 Concurrent Versions System . . . 30
5.3.1 Optimistic Concurrency Control . . . 31
5.3.2 Client-Server Architecture . . . 32
5.3.3 Project Versioning Granularity . . . 32
5.4 Subversion . . . 33
5.5 Towards Distributed Version Control . . . 33
5.5.1 Collaboration Workflows . . . 36
5.6 Git . . . 37
5.6.1 Branching Philosophy . . . 37
5.6.2 Full Copy Object Storage . . . 37

5.6.3 Object Integrity and Identity . . . 38
5.6.4 Versioning Model . . . 38
5.6.5 Rebasing . . . 39
5.6.6 Other Important Concepts . . . 39
5.7 Mercurial . . . 40
5.7.1 Versioning Model . . . 40
5.7.2 Differences from Git . . . 41
5.8 Summary . . . 42

6 The Hydra Approach to Versioning 43
6.1 Logical Units . . . 43
6.1.1 Conceptual Introduction . . . 43
6.1.2 Logical Units in Alpha-Flow . . . 44
6.1.3 Logical Units in Software Development . . . 45
6.1.4 Definition of Logical Units . . . 48
6.1.5 Benefits of Logical Units . . . 51
6.2 Version Validity . . . 53
6.2.1 Conceptual Introduction . . . 53
6.2.2 Validity in Alpha-Flow . . . 54
6.2.3 Validity in Software Development . . . 55
6.2.4 Definition of System and Valid Path . . . 57
6.2.5 Benefits of Validity Tracking . . . 58
6.3 Non-Conventional Means of Support . . . 59
6.3.1 Logical Unit Support . . . 59
6.3.2 Valid Version and Path Support . . . 62
6.4 Alpha-Flow Adequacy . . . 65
6.5 Summary . . . 65

7 Design – Versioning Core 67
7.1 Versioning Core . . . 67
7.1.1 Versioning Model . . . 67
7.1.2 Repository Design . . . 70
7.1.3 Versioning Example . . . 77
7.2 Multi-Headed Versioning . . . 79
7.2.1 Extension of the Versioning Core . . . 79
7.2.2 Repository Design . . . 83
7.2.3 Multi-Headed Integration Commits . . . 85
7.3 Validity Tracking . . . 89
7.3.1 Property vs. Path Based Validity . . . 89
7.3.2 State Validity Extension . . . 90
7.4 History Manipulation . . . 91
7.4.1 Insert and Temporary Commits . . . 91
7.4.2 Fingerprint Addressable Storage . . . 91

7.5 Summary . . . 92

8 Design – User Interfaces and Subsystems 93
8.1 User Interface . . . 93
8.1.1 Commands . . . 94
8.1.2 Command Line Interface . . . 95
8.1.3 Graphical User Interface . . . 99
8.2 Persistency Subsystem . . . 100
8.2.1 Components . . . 100
8.2.2 Terms . . . 101
8.2.3 Functionality . . . 102
8.2.4 Data Access Objects . . . 103
8.2.5 Configuring the System to the User . . . 104
8.3 Logging Subsystem . . . 105
8.3.1 Logging Levels . . . 106
8.3.2 Logger Design . . . 106
8.4 Differential Calculation . . . 106
8.4.1 Abstraction Layer . . . 107
8.5 Summary . . . 108

9 Design – Alpha-Flow Integration 109
9.1 Roles . . . 109
9.1.1 Historian Role Definitions . . . 109
9.1.2 Mapping Historian Roles to Version Control Systems . . . 110
9.2 Alpha-Flow Interface Design . . . 110
9.2.1 Mapping Alpha-Flow Requirements to Abstract Roles . . . 111
9.2.2 Interface Definition . . . 111
9.3 Summary . . . 111

10 Implementation 113
10.1 An Agile Approach . . . 113
10.1.1 Iterations . . . 113
10.1.2 Task Definition and Execution . . . 113
10.1.3 Assessment . . . 114
10.2 Versioning Core . . . 114
10.2.1 Fingerprint Calculation . . . 114
10.2.2 Maintaining Configuration . . . 116
10.3 User Interface . . . 118
10.3.1 Command Regular Expressions . . . 118
10.3.2 GUI Visualization . . . 119
10.4 Subsystems . . . 120
10.5 Alpha-Flow Integration . . . 120
10.6 Summary . . . 120

11 Assessment **121**
11.1 Overall Assessment . . . 121
11.2 Software Metrics . . . 121
11.3 Functionality Evaluation . . . 123
11.4 Performance Evaluation . . . 123
11.4.1 Test Benchmark . . . 123
11.4.2 Test Plan and Execution . . . 123
11.4.3 Test Results . . . 124
11.4.4 Assessment . . . 124
11.5 Future Work . . . 125
11.5.1 Maturity Work . . . 125
11.5.2 Conceptual Work . . . 125
11.6 Summary . . . 126

12 Conclusion **127**

Appendices **129**

A Hydra – Quick Start **131**
A.1 Installation . . . 131
A.1.1 Organization . . . 132
A.1.2 Shell Script . . . 132
A.2 Starting Hydra . . . 132
A.2.1 Execution Modes . . . 132
A.2.2 Initializing A New Repository . . . 133
A.2.3 Other Parameters . . . 133
A.3 Creating and Managing Logical Units . . . 133
A.3.1 Stage and Logical Units . . . 134
A.4 Dealing with Files . . . 134
A.4.1 Listing Directory/File Contents . . . 134
A.4.2 Adding and Removing Files . . . 134
A.4.3 File Differentials . . . 134
A.5 History Management . . . 135
A.5.1 Committing a Version . . . 135
A.5.2 Reverting the Workspace . . . 135
A.5.3 Logging a History . . . 136
A.6 System Configuration and Commands . . . 136

Hydra – Command Cheatsheet **138**

Bibliography **III**

List of Figures

1.1 Alpha-Doc Structure 3

4.1 Alpha-Episodes and Collaborating Healthcare Professionals 17
4.2 Alpha-Card Model 18
4.3 Alpha-Flow System Architecture 19
4.4 Synchronization Anomalies 20
4.5 Example Version Vector 21
4.6 Version Vector Detection of Anomalies 21
4.7 Local Reconciliation of Concurrent Changes 22
4.8 History Depiction 22
4.9 alpha-VerVarStore Persistence Structure 23

5.1 SCCS Delta Visualization 26
5.2 SCCS and RCS Architecture and Workflow 27
5.3 Forward vs. Reverse Delta Visualization 28
5.4 Version and Variant Difference 29
5.5 Optimistic Concurrency [Gru86] 32
5.6 CVS and SVN Architecture and Workflow 32
5.7 Distributed VCS Architecture and Workflow 35
5.8 Integration Manager Collaboration Workflow 36
5.9 Git Versioning Model [Muk05] 39
5.10 Git Rebasing 40
5.11 Mercurial Versioning Model [Muk05] 41

6.1 Conceptual Logical Units 44
6.2 Alpha-Flow Mapping to Conceptual Logical Unit Venn Diagram 45
6.3 Software Project Structure and Possession 48
6.4 Comparison of Single-Headed and Multi-Headed Versioning Paradigms . 50
6.5 System and Valid Path Initial Concept 54
6.6 Determination of Version Validity 57
6.7 System and Valid Path Visual Definition 58
6.8 Encapsulating Subcomponent Techniques Comparison 61
6.9 Explicit Properties Based Valid Path 62
6.10 Branching Based Valid Path 63
6.11 Blessed Repository Valid Path 64

7.1 Artifact, Container, and State CRC Cards 68

7.2 Artifact, Container and State Role Relationships . . . 69
7.3 Versioning Model Class Diagram . . . 70
7.4 VCS Component Parts . . . 71
7.5 Hierarchical Storage Organization Comparison . . . 73
7.6 Updated Versioning Model Class Diagram Including Fingerprint . . . 74
7.7 Persisted Artifact Format for Calculating Fingerprint . . . 75
7.8 Persisted Container Format for Calculating Fingerprint . . . 75
7.9 Persisted State Format for Calculating Fingerprint . . . 76
7.10 Example Directory Structure and Fingerprints . . . 76
7.11 Versioning Model Format References Example . . . 77
7.12 Versioning Examples Outset Situation . . . 78
7.13 Sharing of References for Unchanged Artifacts . . . 78
7.14 Sharing of References for Moved Artifacts . . . 79
7.15 Logical Unit and Stage CRC Cards . . . 80
7.16 Logical Unit and Stage State CRC Cards . . . 81
7.17 Logical Unit and Stage Role Relationships . . . 82
7.18 Final Versioning Core Class Diagram . . . 83
7.19 Persisted Logical Unit and State Format for Calculating Fingerprint . . . 84
7.20 Persisted Stage and State Format for Calculating Fingerprint . . . 84
7.21 Repository Structure Example . . . 85
7.22 Initial Integration Commit . . . 86
7.23 Second Integration Commit . . . 87
7.24 First Recursive Integration Commit . . . 88
7.25 Second Recursive Integration Commit . . . 88
7.26 Property Based State Validity . . . 89
7.27 Path Based State Validity . . . 90
7.28 State Validity Support Extension . . . 91
7.29 Fingerprint Addressable Storage State Format . . . 92

8.1 User Interface Activity Diagram and Area of Interest . . . 93
8.2 Command Class Diagram . . . 96
8.3 CLI Command Processing Activity Diagram . . . 97
8.4 Command Class Diagram Including CommandRegex . . . 98
8.5 GUI Layout Design . . . 100
8.6 User Interface Class Diagram . . . 101
8.7 Component and Interaction Overview . . . 102
8.8 Store and Retrieve Functionality Visualization . . . 103
8.9 Record and Load Functionality Visualization . . . 103
8.10 Data Access Object Class Diagram . . . 104
8.11 Persistency Subsystem Class Diagram . . . 105
8.12 Logging Subsystem Class Diagram . . . 106
8.13 Differential Interface . . . 107
8.14 ChangeSet and Change Class Diagram . . . 108

9.1 Historian Roles 110
9.2 VCS Responsibilities Mapped to Historian Roles 110
9.3 Alpha-Flow Abstract Versioning Interface 112

10.1 Calculation of Artifact Fingerprint 115
10.2 Calculation of Container Fingerprint 115
10.3 Calculation of State Fingerprint 116
10.4 Configuration Class Diagram 117
10.5 Example Regular Expression Pattern 118
10.6 GUI Screenshot 120

11.1 Lines of Code Distribution 121
11.2 Core Executable Size 122
11.3 Test Coverage Summary 122

List of Tables

5.1 VCS Summary 42

6.1 Granularity Levels Comparison 49
6.2 Alpha-Flow Adequacy 65

8.1 VCS Capabilities and Necessary Extensions 95
8.2 CLI Command Usage 99

11.1 Stress Test Results 124

List of Abbreviations

APA Adornment Prototype Artifact 16
API Application Programming Interface 4
BLOB Binary Large Object 38
CDA Clinical Document Architecture 15
CLI Command Line Interface 95
CM Configuration Management 30
CRA Collaboration Resource Artifact 16
CRC Class-Responsibility-Collaboration 68
CSCW Computer Supported Cooperative Work 14
CVS Concurrent Versions System 30
DAO Data Access Object 100
dVCS distributed Version Control System 37
dDPM distributed Document-oriented Process Management 3
GB Gigabyte 37
GUI Graphical User Interface 99
HL7 Health Level 7 15
IDE Integrated Development Environment 30
IEEE Institute of Electrical and Electronics Engineers 26
I/O Input/Output 104
JAR Java Archive 12
JUNG Java Universal Network/Graph Framework 119
JuRR JUnit Runner and Reporter 122
KB Kilobyte 37
MB Megabyte 119
mVCS Multi-Headed Version Control System 67
NIO New Input/Output 104
OS Operating System 132
PDF Portable Document Format 3

PSA Process Structure Artifact 16
QA Quality Assurance 56
RAM Random Access Memory 124
RCS Revision Control System 27
SCCS Source Code Control System 25
SHA-1 Secure Hash-1 38
SVN Subversion 7
TDD Test Driven Development 8
USB Universal Serial Bus 4
VCS Version Control System 1
XML Extensible Markup Language 3
XP eXtreme Programming 9
UI User Interface 8
UID Unique Identifier 73
UUID Universal Unique Identifier 92

1 Introduction

Version Control Systems (VCSs) are ubiquitous in software development and other fields such as medicine, justice and business, where persistence and tracking of changes to electronic documents, to be referred to as ***artifacts***, is legally required or otherwise advantageous. VCSs facilitate the collaboration of a team, distributed around the world, operating at different times on a common set of files by supporting the concurrent manipulation of the shared files, recording or ***committing*** the changes made to those files and the parties responsible for introducing each change. Additionally, they are able to ***merge*** the changes made by various parties into a coherent version reflecting the result of the concurrent alterations. Finally, they allow the files to be reset or ***reverted*** to a previously persisted state.

Over the last decades the capabilities of VCSs have evolved to suit the demands of their users. Initial VCSs enabled a single user to simply track changes to designated files locally and revert them to a designated previous state. Today, VCSs provide support for multiple distributed users in various topographical organizations along multiple lines or ***branches*** of development. They have become an essential ingredient to any successful software development project; sharing a similar critical status as the implementation language and developmental environment.

However, VCSs lack capabilities in two important areas: (1) subcomponent organization and management and (2) version validity tracking. These shortcomings create a need to employ VCSs in non-standard ways or develop extensive support systems to accomplish common every day tasks faced in software development and other areas they are employed.

Multi-Headed Versioning

Contemporary VCSs are designed to record the evolution of a project as a sequence of changes, commonly referred to as the project's ***history***. Each of these set of changes describes a single transformation of the overall project from one discrete state to another. A recorded state is referred to as ***version***. These systems maintain the reference to a single ***head*** version, which represents the most recently recorded state of the project. Any other version may be navigated to from this referenced version. This approach will be referred to as ***single-headed versioning***.

[0]Side Note: Throughout this thesis a number of terms commonly associated with VCS technology, such as ***artifact***, ***commit***, ***revert*** and ***version***, will be used. These terms will appear in an bold-italicized font where they are first introduced. The Glossary will also provide a brief referential definition.

However, most software development efforts are broken into logically independent and decoupled subprojects. Each being designed and developed generally independent, perhaps by completely different teams of developers. Even though software engineers has long ago abandoned attempts to compose software as a single monolithic component, VCS technology has never evolved to accurately reflect the decomposition of the overall project into a set of subprojects. This creates a conceptual gap between the VCSs and their primary field of employment.

To support independent subprojects, VCSs must be extended to manage multiple heads; one for each subproject and one for that represents the compositional state of the overall project. This approach will be referred to as ***multi-headed versioning***. No VCS, known to the author, explicitly supports the independent management of each subproject's history while simultaneously and independently maintaining the history of the overall project.

Version Validity

Each recorded version is considered equally valid within contemporary VCSs. However, versions may exhibit inherent qualities, such as being fault-free or non-build breaking, that differentiate their acceptability and impact on the forward progress of the overall development. Based on the context of the application a version may be classified as either acceptable and valid or unacceptable and invalid.

Unknowingly working from an invalid or faulty version increases the difficulty of producing error-free software. A system that is capable of differentiating versions based on their validity characteristics would improve the quality of software developed by ensuring that all changes are based on an acceptable state.

Validity is the application-defined characteristic of a version that expresses its correctness or acceptability. No VCS, known to the author, is capable of differentiating versions based on its inherent validity properties or is capable of managing a history that describes the forward progress of development.

1.1 Project Motivation

In this section, the motivations driving this project will be investigated. The α-Flow project will be introduced and provides a concrete example of a system that benefits from the introduction and support of these two concepts.

1.1.1 Alpha-Flow Project Overview

The previously mentioned challenges are not restricted to the field of software development, but may be seen in any number of other areas. α-Flow is one example of a project that benefits from the employment of these concepts. In this section the α-Flow project will be introduced and a summary the key characteristics relevant to this thesis

will be presented. A more thorough analysis of α-Flow is found in Chapter 4: *Analysis – Alpha-Flow* on page 13.

α-Flow, as described in [NL09], [NL10] and [NSWL11], is a distributed Document-oriented Process Management (dDPM) system, which aims to support the inter-institutional healthcare treatment process. It employs a workflow that mimics the traditional paper-based workflow currently common in healthcare. In order to treat a patient within the paper-based approach, a collaborating healthcare professional receives a copy of a case file; often hand-carried by the patient. Inside each case file is a set of documents that represent the information gathered from handling the patient by other collaborators and a request for additional information.

Within α-Flow, this distributed case file is represented as a replicated and synchronized electronic document, an α-Doc. An α-Doc encapsulates a collection of subcomponents and is imbued with active properties [LED+99] that enable it to interact with the user and other documents. The subcomponents, α-Cards, represent the individual files contained within the case file. Their collective state represents the state of a patient's treatment and the overall situational awareness shared by all collaborating healthcare professionals. An action or information may be requested by creating an α-Card and fulfilled by updating the α-Card with the requested information or result of the action. The sequences of changes applied to the α-Cards represents the treatments progress and a treatment is considered complete when no further information remains to be exchanged. The treatment process itself is abstractly considered an α-Episode and represents the communal goal of treating a patient.

Each α-Card represents a specific task to be completed in a patient's treatment and is composed of two files: a (1) descriptor and (2) payload. The descriptor, an Extensible Markup Language (XML) file, maintains all process relevant status attributes and is used to steer the collaborative workflow. The payload represents the result of the task and may take the form of any electronic document, such as Microsoft Word or Portable Document Format (PDF). Figure 1.1 depicts the general structure of an α-Doc.

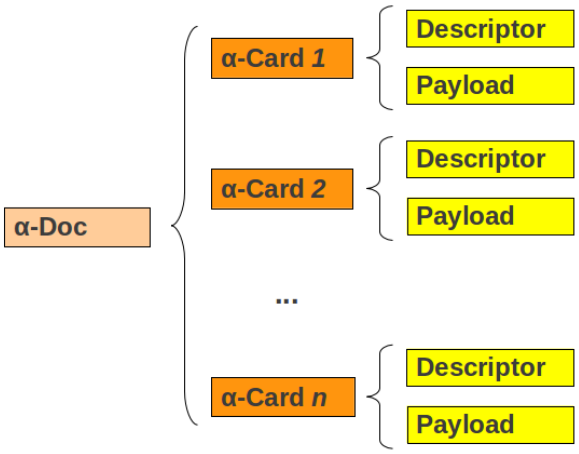


Figure 1.1: Alpha-Doc Structure

1.1.2 Alpha-Flow's Key Characteristics

The following is a list of the key characteristics of α-Flow that are relevant to this thesis:

Peer-to-peer infrastructure α-Flow is designed as a loose peer-to-peer architecture. Each α-Doc represents an autonomous peer node and interacts with every other peer α-Doc node of the same α-Episode. Each node maintains a replication of all case data. New nodes are created by simply creating a copy of an α-Doc and are automatically integrated into the peer network.

Lightweight Application The entire α-Doc should be a standalone application in the form of a single document that may be exchanged or moved from location via email or Universal Serial Bus (USB) stick. Therefore, its memory signature must remain as small as possible.

Heterogeneous Systems α-Flow is designed to bridge the inter-institutional gap and operate on heterogeneous systems. No assumptions about the underlying system or its architecture may be made. Any supporting system must also be platform independent or at least provide support on most common operating systems. It must have few or no system dependencies and no installation requirements.

Java implementation α-Flow inherits much of its platform independence from its Java implementation. As such, any supporting system must be implemented in or be capable of being programmatically accessed through Java.

Messaging-based Synchronization Since it can never be guaranteed that two α-Docs are simultaneous available for direct exchange of data, a *store-and-forward* approach is used. Data is exchanged between α-Docs asynchronously and may arrive delayed, out-of-order or not at all. An internal algorithm examines the arriving messages, determines its logical position within the history and determines if the new data creates a global conflict which must be resolved.

No human Interaction for Versioning All aspects of the system's versioning is programmatically controlled by an internal rules-based engine. The versioning system, as an embedded subsystem, should never create a conflict state that requires direct human interaction with the VCS. Any necessary human intervention will be handled by the α-Flow engine, accessing the embedded VCS through its Application Programming Interface (API).

Binary or Proprietary Data Formats The greater mass of data under version control is non-text or binary data, such as an x-ray scan, XML or PDF.

1.1.3 Multi-Headed Versioning

Three aspects of α-Flow present a challenge for the conventional single-headed versioning paradigm: (1) the α-Doc's inherent compositional structure, (2) actor and α-Card

autonomy, (3) specified ownership per α-Card. All of these are caused by the coarse level of ***versioning granularity***, i.e. the encapsulation of the entire project's state into a single version, used in contemporary VCSs which cannot account for the independent nature of the α-Doc's α-Cards.

Compositional Structure

Dealing with complexity is not the only reason for decomposing a project into a set of logically independent subcomponents. α-Flow is an example of a project whose versioned data, the α-Doc, is naturally described as a composition of relatively independent units, the α-Cards.

Conventional single-headed versioning is unable to represent this compositional structure. It is only capable of representing a single all encompassing state. Therefore, it cannot describe a version of an α-Doc as a composition of α-Card versions.

Employing the multi-headed versioning paradigm allows the compositional state of the α-Doc to be more accurately portrayed. The history of each α-Card is maintained by its respective independent head and the α-Doc's state is represented as the composition of the α-Card heads.

Autonomy

Each α-Card represents a single loosely independent interaction or treatment involving a medical professional and a patient. In order to reflect the dynamics of the treatment episode correctly, the α-Cards must also be managed independently and have the same level of autonomy as their respective actors (i.e. the medical professionals). Any restriction to the autonomy of the α-Cards restricts the autonomy of the actors.

Single-headed versioning restricts this autonomy because the state of each α-Card is intrinsically tied to the state of each other α-Card. Any changes made to one α-Card result in a change of state for another α-Card as they are ultimately connected through the single all-encompassing state produced by this versioning paradigm. Thus the actions of one actor are directly impact another actor and are not autonomous.

Multi-headed versioning allows each actor to autonomously alter and record changes to an α-Card without influencing another actor. This is because each α-Card has its own head version which maintains the changes instead of all α-Cards synchronizing over a single shared head.

Ownership

Each α-Card has a specified owner or medical professional responsible for the accuracy of the information it presents. Committing updates to an α-Card is similar to the doctor's signature on a medical report.

Single-headed versioning cannot reflect the specified ownership per α-Card. Any commit made by an actor encompasses the entire α-Doc and cannot be localized to the specific α-Card for which they are responsible. This would be equivalent to a doctor

signing all medical documents within the case file whenever they made a change. This confuses the allocation of authority and responsibility and breaks the verifiability of the system.

Multi-headed versioning is capable of assigning ownership to each logically independent α-Card. Any commit made to an α-Card is restricted to the contents of that specific α-Card. This is equivalent to a doctor signing only the documents for which they are responsible.

1.1.4 Version Validity

The validity of a version within α-Flow plays a critical role. The health of the patient depends on the doctor making well founded decisions based on valid or correct information. Doctors basing their decisions on invalid information could clearly risk the health of the patient.

Therefore, it is critical to avoid the use of invalid information. To accomplish this, each version of information must be labeled as being valid or invalid and only valid information must be presented to the user of the system.

1.2 Project Goal

The goal of this project is to design and implement a VCS employing the concepts of the multi-headed versioning and version validity tracking. This system will be able to fulfill the versioning requirements of the α-Flow project, to be formulated in Chapter 3: *Requirements* on page 9. It will be able to: (1) track the changes made to each α-Card independently, (2) differentiate between versions based on their validity, (3) integrate into the α-Flow project and (4) be programmatically controlled.

2 Method

First, an initial analysis of the product's requirements was conducted through the use of an inception deck. The critical requirements beyond basic versioning functionality are: (1) support for independent α-Card versioning, (2) differentiation of versions based on their validity and (3) a minimalistic implementation tailored to meet only the specified needs of the α-Flow system. The project's requirements were continually refined and prioritized to represent the most current perspectives of the client. The requirements and their management are presented in Chapter 3: *Requirements* on page 9.

Once an initial set of requirements was derived, the α-Flow project was analyzed to gain a better understanding of the purpose of the requirements and how version control is perceived and integrated into the system. Of special interest were the business logic's rules, the α-Doc synchronization process and the versioning subsystem. Chapter 4: *Analysis – Alpha-Flow* on page 13 provides a summary of this effort's findings.

Various VCSs were then analyzed for their suitability for extension and employment in the α-Flow system. The primary VCSs analyzed were Subversion (SVN)[1], Mercurial[2] and Git[3]. Chapter 5: *Analysis – Version Control Systems and Their Evolution* on page 25 provides a summary of important VCSs and their role in the evolution of VCS technology.

Then the conceptual description of multi-headed versioning and version validity was considered. Once these concepts were defined, each of the analyzed VCSs were considered with respect to their appropriateness. However, no VCS suitable for supporting α-Flow system was found. This drove the decision to design and implement a new VCS. Chapter 6: *Impetus of Hydra* on page 43 provides the definitions for logical units and validity needed to support the new versioning concepts and describes the inadequacy of contemporary VCSs with respect to these concepts.

Next, the focus was turned to designing a prototypical system to support α-Flow's versioning needs. Much of the design for the basic versioning functionality was pulled from Git. Its peer-to-peer architecture is best suited for the α-Flow's distributed infrastructure and its development is well documented and open source. However, the primary design effort was focused on the extension of the system beyond the typical VCS's capability. It was oriented to solve the specific needs of the α-Flow system instead of trying to create a system that only provides the same capabilities as current systems. Attention was focused on supporting logical independence of the α-Cards versioning and defining and manipulating state validity. The system's core design is described in Chapter 7: *De-*

[1]Subversion Homepage: `http://subversion.apache.org`
[2]Mercurial Homepage: `http://mercurial.selenic.com`
[3]Git Homepage: `http://git-scm.com`

sign – Versioning Core on page 67. A User Interface (UI) and various subsystems, such as Persistency and Logging, were also designed to support the versioning core. Chapter 8: *Design – User Interfaces and Subsystems* on page 93 provides a detailed coverage of these topics. Additionally, the integration into α-Flow was designed. See Chapter 9: *Design – Alpha-Flow Integration* on page 109 for more details.

The developmental effort was accomplished in four iterations organized in the same manner as the design: (1-2) Versioning Core (two iterations), (3) UIs and Subsystems and (4) α-Flow Integration. For each of these iterations, a set of features to be implemented were agreed upon and the details satisfying their acceptance were specified. A non-distinct agile method based primarily on Kanban, Scrum and Test Driven Development (TDD) was employed during the development. Weekly meetings were held with the client to discuss the work done and clarify any outstanding questions for the next week's work. For a more in depth discussion of the approach to development see Section 10.1: *An Agile Approach* on page 113.

All iterations proceeded as expected. In the first and second iterations the versioning core was developed. In the third iteration the UIs and various subsystems, such as the persistence subsystem and logging system, were implemented. The fourth iteration integrated the developed system into α-Flow. The majority of the information on these topics is covered in their design and will not be redundantly presented. However, a selection of interesting portions, including configuration management and regular expression command parsing, is presented in Chapter 10: *Implementation* on page 113.

Finally, the project was evaluated and recommendations for future improvements for the system were provided. The product was evaluated against basic software metrics, performance and ability to provide the functionality required by the α-Flow system. Additionally, a series of future work, covering both system maturity and conceptual aspects, was considered. The project's evaluation is presented in Chapter 11: *Assessment* on page 121.

3 Requirements

In this chapter the process in which the project's requirements were gathered and managed will be first discussed. Next, the functional requirements will be defined; followed by the project's constraints or non-functional requirements. Finally, a summary will be provided of the key points discussed in this chapter.

3.1 Requirements Management

An blended mixture of agile developmental methods, drawing mostly from Scrum [Coh10] and eXtreme Programming (XP) [Bec99], was employed during this project in order to derive the requirements. An initial project inception meeting provided the initial project's orientation, goals and rough set of requirements. These requirements were then refined and prioritized. A set to be fulfilled was chosen during the first iteration's planning meeting.

The requirements being actively developed were added to the sprint backlog during the planning meeting and the other discovered requirements were added to and maintained in the product backlog. Prior to each iteration, the planning meeting was used to define the scope of work for the coming iteration. Weekly huddles were used to clarify any questions and to introduce new requirements that were added to the project backlog. In order to provide an increased flexibility during the development and simplify progress tracking, sets of logically relating requirements were typically generalized into features.

3.2 Functional Requirements

This section will define the project's functional requirements and their driving intent based on the needs of the α-Flow project. Each of these requirements will be satisfied by the implemented system. The success of this prototypical implementation will be judged by the satisfaction of these requirements.

3.2.1 Basic Version Control Capabilities

The system must be able to provide the basic versioning capabilities. It must be able to persist or commit the current state of multiple artifacts, return the artifacts to a previously persisted state and provide a description of the set of changes that have been made those artifacts. Merging of concurrent alterations is not required to be supported

as α-Flow operates primarily on binary data, such as x-rays, and other non-mergable formats.

The system should be a functional versioning system without providing branching and merging capabilities or other functionality. This requirement will provide support for the basic versioning needs for α-Flow.

3.2.2 All Versions of Artifacts Must be Maintained Locally

The system must maintain all versions of an artifact within a local repository. Changes made in one repository, must be able to be retrieved and integrated within another repository. Direct exchange between the repositories is not necessary, data exchange will be supported by the α-Flow system. Local maintenance of all versions is necessary because of α-Flow's loose peer-to-peer architecture. It can never be assumed that peers are simultaneously available online to support direct synchronization and thus all versions must be maintained locally to support the autonomy of each peer.

3.2.3 Alpha-Doc and Alpha-Card Versioning

The changes to each α-Card and the encapsulating α-Doc must be independently tracked and each element must be able to support querying of its individual history. Each α-Card currently is statically composed of two artifacts: descriptor and payload. However, the system should be developed in a manner that allows the α-Card to be dynamically defined to include an arbitrary set of artifacts. Additionally, the overall state of an α-Doc's composing α-Cards must also be maintained and reflect a coherent set of α-Card interdependencies.

This requirement will support the concept that each α-Card within an α-Doc is an independent logical unit and must be tracked as such. The overall α-Doc state describes state of the treatment process, the overall situational understanding and maintains the loose interdependencies between the individual α-Cards. This is the primary desired functionality that is not available in other VCSs.

3.2.4 Version Differentiation Based on Validity Characteristics

The system must maintain each version's system-defined validity characteristic and provide means to return only valid versions. The system must be able to navigation through an α-Card's valid versions while ignoring any invalid versions. This is important because basing medical decisions unknowingly on invalid information could endanger a patient.

3.2.5 Support for Alpha-Flow Global Conflict Resolution Schemes through History Manipulation

α-Flow employs a internal rules engine to detect and resolve global conflicts stemming from concurrent manipulation of shared artifacts and asynchronous communication.

Due to its asynchronous communication, the system must be able to deal with data arriving out-of-order, delayed or not at all. The system uses logical timestamps to determine the correct evolutionary sequence and resolve global conflicts. For a more in depth introduction to the system's detection and resolution strategies, see [Wah11] or the summary provided in Section 4.4: *Off-Line Synchronization* on page 20. These strategies require a number of atypical operations to support its functionality. The versioning system must be able to:

- maintain the defined validity of versions
- identify versions as temporary
- update previously committed versions with new information
- insert/reorder versions
- allow traversal of versions along the valid evolutionary paths
- support querying of information about specific versions

This requirement will support the asynchronous and distributed nature of the α-Flow system, where human interaction and network delays may cause an out-of-order arrival of communication messages.

3.2.6 Produce a Visual Depiction of Histories

The system must be able to provide a visual depiction of each α-Card's and the α-Doc's histories. This depiction must differentiate between valid and invalid states. This provides a simple method of analyzing the flow of an interaction that the α-Doc's history describes.

3.3 Non-Functional Requirements and Constraints

This section will describe the non-functional constraints on the versioning system derived from α-Flow. Most of these requirements can only be subjectively assessed but their satisfaction will also influence the success of this prototypical implementation.

3.3.1 No Human Interaction

This system must successfully operate without human intervention of any kind. It must be programmatically controlled and execute automatically based on the rule-based triggering of events.

3.3.2 Platform Independence

This system must be platform independent in order to support its employment in the expected heterogeneous institutional environments. It must not make any assumptions

about the underlying environment or require installation of supporting software.

3.3.3 Java-Based Implementation

The system must be implemented in Java and consolidated into a Java Archive (JAR) file. The functionality must be accessible over a simple to use Java API. This will allow the system to easily integrated into the α-Flow project.

3.3.4 Maven Project Integration

The system must be incorporated into the α-Flow's Maven[1] build environment and produce and install the necessary JARs into each developer's local Maven repository. However, it must remain an independent project that is loosely coupled to the α-Flow project over the predefined alphaVVS interface. The interface may be altered as needed, but should be abstracted to the level where it may be used to employ other VCSs.

3.3.5 Lightweight

The system must remain lightweight as possible since it will be integrated into a larger package which must remain agile. Any additional features supported by the system, which are not directly required by α-Flow, must be able to be stripped to remove excess size.

[1] Maven Homepage: `http://maven.apache.org`

4 Analysis – Alpha-Flow

This chapter will provide a summary of the α-Flow system's fundamental concepts, domain model, and system architecture. In Chapter 5: *Analysis – Version Control Systems and Their Evolution* on page 25, VCS technology evolution is described and in Chapter 6: *Impetus of Hydra* on page 43 they are assessed with regards to their appropriateness for supporting α-Flow's versioning needs.

4.1 Fundamental Concepts

α-Flow, as described in [NL09], [NL10], [TN11], [NSWL11], and [NL12] is a distributed Document-oriented Process Management (dDPM) system, which aims to support the inter-institutional healthcare treatment process. The concepts of the case handling paradigm and document-centered collaboration provide the foundational framework on which the α-Flow project is built. These concepts will first be introduced to provide a frame of reference for the description of α-Flow's domain model and architecture.

4.1.1 Case Handling Paradigm

The intention of α-Flow is to employ a workflow similar to the traditional paper-based workflow currently common in healthcare. Within the paper-based approach a collaborating healthcare professional receives a copy of the distributed case file, often hand-carried by the patient. Inside each case file is a set of documents that represent the information gathered from handling the patient by other collaborators and a request for additional information. Requests for more information typically assume the form of a request voucher and information gathered is presented typically in the form of a report, such as an x-ray or lab report. Therefore, every document within the distributed case file is the result of an activity.

To provide a similar methodology for supporting this collaboration, α-Flow employs the case handling paradigm. In the case handling paradigm [vdAWG05] the distributed case file assumes the central responsibility for controlling the workflow. Unlike traditional workflow management techniques, this paradigm attempts to assist the decision making process of collaborating participants instead of predefining process steps. The state and structure of any case is defined by the presence or absence of data objects. Case handling's core features are [vdAWG05]:

- *... provide all information available (i.e., present the case as a whole rather than showing just bits and pieces),*

- *decide which activities are enabled on the basis of the information available rather than the activities already executed,*
- *separate work distribution from authorization and allow for additional types of roles, not just the execute role,*
- *allow workers to view and add/modify data before or after the corresponding activities have been executed (e.g., information can be registered the moment it becomes available).*

Within the traditional healthcare workflow, these data objects may be considered the set of files within the case file. α-Flow assumes that the progression of the treatment, or status, can always be represented by the state of the documents within the case file. Through this concept, the workflow is not described as a set of activities rather through the set of documents. The process' progress is represented through the sequential introduction of new files and changes to previously introduced files. The state of the set of documents in the case file represents the overall situational understanding of a patient's treatment at that point in the process.

4.1.2 Document-Centered Collaboration

Support for inter-institutional collaboration of participating healthcare professions operating asynchronously in heterogeneous environment presents a significant challenge. Each participant must remain autonomous, but means to support their collaboration must be provided. The concepts of document-centered collaboration provide α-Flow with an infrastructure that may be employed to support the collaborative nature of its distributed case file notion.

As described in [LED+99], Computer Supported Cooperative Work (CSCW) must provide support for content work and support for coordination. In the described document-centered collaboration, coordination and collaborative functionality becomes an aspect of the artifact instead of the application.

In the document-centered collaboration approach, documents are enabled with an extended ability to interact with a user. An active document associates behavior to a document through the use of active properties. Active properties are executable code fragments, which represent a specific computational act and may be triggered by some event such as the reading, writing, moving or deleting of the document. The behavior defined within the active code may be used to take an appropriate action, such as sending notifications or preventing the triggering event. The association of active properties with a given active document imbues it with the generalized ability to respond to a given situation and interact with its user.

Associating content with behavior allows a developer to bind the collaborative application's semantics directly to the content on which they operate. Active properties also have the ability to query the environment external to the document, providing them with an extended situational awareness they may use to adjust their behavior.

On important side effect of this approach is the actual document that is enabled with active properties remains unchanged. The document may still be accessed and manipulated through any appropriate application, such as MS Word or Excel.

4.2 Domain Model

α-Flow's domain model represents a patient's inter-institutional treatment process as an α-Episode. Similar to the case handling paradigm's product, described in [vdAWG05], the α-Episode represents the common goal of all collaborators to treat a patient's condition.

4.2.1 Alpha-Docs

Each α-Episode is electronically realized as an α-Doc, which may be visualized as a distributed case file that is virtually shared by all collaborating healthcare professionals. An α-Doc is an active document, as described in [LED+99], imbued with the abilities to exchange information with other peer replicates and interact with users. Whereas the α-Episode represents the overall process shared by all collaborators, each α-Doc provides an autonomous means of interaction for a designated human actor and emphasizes the artifact dimension of the case file.

The required set of collaborators may not be initially definable as little may be known about the patient's condition. New collaborators may be identified based on their area of expertise and the demand for more information in their area of expertise. These new collaborators may be integrated into the treatment scenario by receiving a copy of an α-Doc. Therefore, the α-Doc is an autonomous unit of information exchange that is synchronized with all other peer copies and maintains all shared case related information.

4.2.2 Alpha-Cards

Whereas the α-Doc represents the distributed case file, each of the separate documents maintained in the distributed case file are represented within α-Flow as an α-Card. α-Docs are thus composed of a set of α-Cards. α-Cards in turn provide organizational accountability, validity and are the subject of atomic synchronization actions.

Each α-Card represents a specific task to be completed in a patient's treatment and is composed of two subcomponents: the descriptor and payload. The descriptor maintains all process relevant status attributes, referred to as α-Adornments, and are used to steer the collaborative workflow. The payload represents the result of the task and may take the form of any electronic document, such as a Word document, PDF or Health Level 7 (HL7) Clinical Document Architecture (CDA). Each task in a patient's case is planned by creating a descriptor and it is fulfilled by introducing the report or other document resulting from the task's execution as the payload. As described in the case handling

paradigm, the treatment process and its state is defined by the creation and changing of α-Cards.

Content and Coordination

The preceding description of an α-Card corresponds to the concept of the traditional paper-based method employed in healthcare. This accounts for the content aspect, but does not account for the coordination aspect needed to support CSCW. α-Flow strictly separates process related data from medical content in order to decouple these two aspects. This leads to the classification of α-Cards into two categories: content and coordination. Content α-Cards are those that deal with medical information as described in the previous section. Coordination α-Cards provide the contextual information necessary to support the inter-institutional collaboration.

Each α-Doc currently maintains three coordination cards: the Process Structure Artifact (PSA), the Collaboration Resource Artifact (CRA) and the Adornment Prototype Artifact (APA). The PSA manages the workflow schema and maintains a list of all content α-Cards and their interdependencies. The CRA manages the processes participants and maintains a list of the participants and how they may be electronically reached (i.e. their email address) in order to support the system's synchronization needs. The Adornment Prototype Artifact (APA) manages the α-Adornment model, described in Section 4.2.3: *Alpha-Adornments* on page 16, used within a given context and allows the set of attributes maintaining the process' status to be dynamically altered to reflect the needs of the collaborative work.

One important difference between content and coordination α-Cards, beyond their intended purpose, is their concept of ownership. Each content α-Card has a specific participant that represents the party responsible for the content and is capable of altering the content. This aspect provides for accountability within the system similar to a doctor's signature on a medical document. However, coordination α-Cards provide a communal context of the process and must be free to be manipulated by any participant. This introduces the possibility of conflicting concurrent changes to be made to the same artifact.

4.2.3 Alpha-Adornments

As described in [NSWL11], α-Flow employs an evolutionary α-Adornment model to mange an arbitrary set of attributes, which provide a flexible means for describing the status of the treatment process. This α-Adornment model is maintained as a prototype by the APA and is cloned when a new α-Card descriptor is created. This prototype generally contains at the least a basic set of attributes commonly used within the system to represent the life-cycle and state of an α-Card, as described in [NL09] and summarized in [NSWL11]. The commonly used adornments include: contributor (owner), object under consideration (patient), validity, visibility, version, variant, syntactic payload type, fundamental semantic payload type, and domain-specific semantic payload type.

4.2.4 Domain Model Summary

α-Flow attempts to mimic the traditional paper-based collaboration process common in healthcare practice today. An α-Episode represents the overall treatment process for a specific patient and defines the communal goal of treating the patient for a specific condition. Therefore, more than one α-Episode may be associated to a single patient. The α-Doc is α-Flow's virtual realization of the treatment's distributed case file. Each involved collaborator receives a peer copy of the α-Doc, which serves as the unit of informational exchange and point of autonomous interaction for each collaborator. Each α-Doc is automatically synchronized to ensure that each collaborator maintains a common situational awareness of the process.

Figure 4.1 depicts a scenario where three treatment episodes occur to handle two patients. The first two episodes are associated with the treatment of specific conditions of patient one where episode two occurs in parallel to episode one. The third episode deals with the treatment of a second patient. Throughout these three episodes, three collaborating healthcare professionals are involved. *Doctor X* and *Doctor Y* are involved with treating the first patient's first condition and *Doctor Z* handles the patient's second condition alone. Treatment of the second patient's condition involves the collaboration of all three doctors.

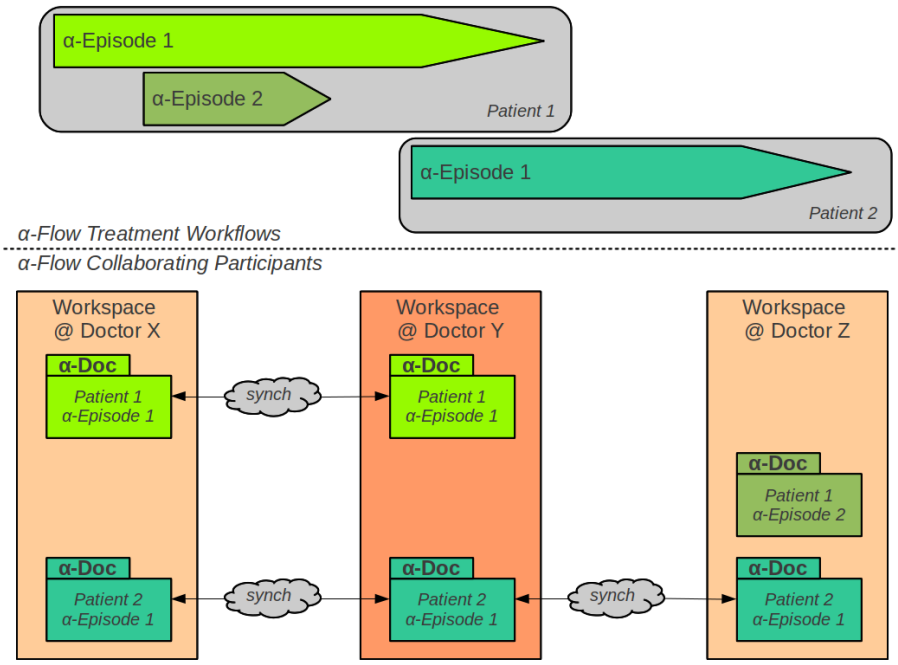


Figure 4.1: Alpha-Episodes and Collaborating Healthcare Professionals

Each α-Doc represents a peer replicate of the data reflecting the status of a treatment episode. The α-Doc is an active document that provides the means to interact with the user and other peer α-Docs. This allows the coordination logic to be embedded into the α-Doc and removes the need for an extensive collaborative network from being

established. An α-Doc is composed of two types of α-Cards: coordination and content. The content cards represent medical data gained throughout the process and are owned by a single collaborator and shared with all peer α-Docs. The coordination cards provide the context for the treatment's collaborative effort and are coequally owned by all collaborators. Each α-Card is composed of a descriptor, which maintains process status attributes, and a payload, which describes the result of an activity.

The described model of peer α-Cards is depicted in Figure 4.2. In this scenario two collaborators, X and Y, share a common view of the treatment's progress through the peer α-Cards. X owns two α-Cards, R^A and RV^M, and possesses a replica of the α-Card, RR^M, which is owned by Y. The coordination cards, PSA, CRA and APA, are owned by both X and Y.

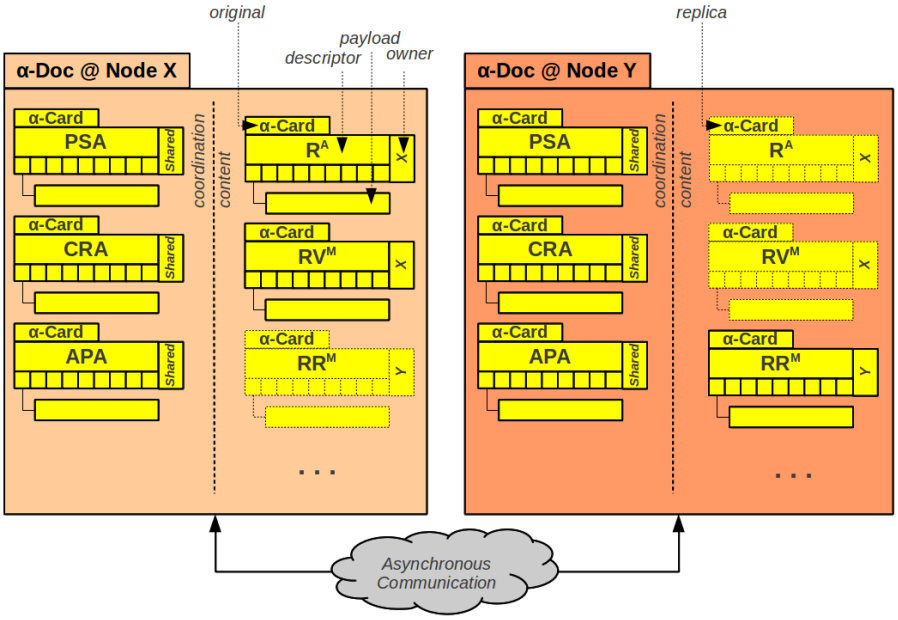


Figure 4.2: Alpha-Card Model

The treatment process and its state is defined by the creation and changing of α-Cards. Information or an activity is requested through the creation of a descriptor and fulfilled by the introduction of the activities result as a report in the payload. In this manner, the progress of a treatment may be described through the changes to the α-Cards.

4.3 System Architecture

α-Flow, as described in [Wah11], is realized as a number of software components, which can be conceptually assigned to one of three different layers, similar to the common three-tier architecture:

- Presentation Layer (user interaction), which is responsible for presenting the in-

formation represented by the α-Doc to the user and providing a mechanism for human interaction.

- Logic Layer (core system logic), which is responsible for managing the various aspects of the domain model and applying the business rules of α-Flow in response to occurring events.
- Data Layer (data persistence and communication), which is responsible for the persistence of the process artifacts and their versions and the communication between peer α-Docs.

The alpha-Startup component is responsible for the initialization of the system core when an α-Doc is opened. The system core consists of the alpha-Startup, alpha-Injector, alpha-Properties and alpha-Adaptive components. The alpha-Injector component provides the capability to introduce new α-Cards into the α-Doc. The alpha-Properties component plays a central role in the system and provides the system's business logic. The alpha-Adaptive component provides support for the evolutionary adaptive attribute model employed to manage the descriptors adornments. Figure 4.3 is a visual depiction of the described key components of the α-Flow system. The dashed lines indicate the interactions involved when initializing the system and the solid lines indicate the interactions after the system has been initialized.

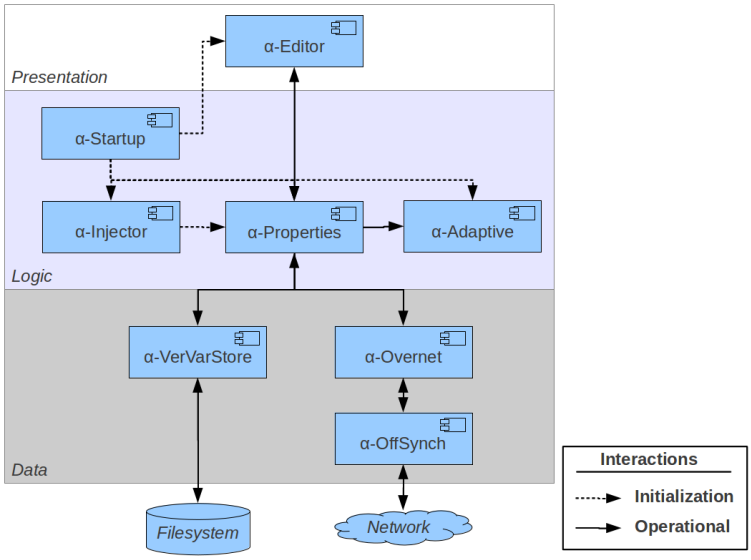


Figure 4.3: Alpha-Flow System Architecture

If the α-Doc has not previously been opened, the alpha-Startup will automatically create an initial α-Card through the functionality provided by the alpha-Injector. Otherwise, the previously created and stored α-Cards will be loaded. Once the system core has been initialized, the alpha-Editor is also initialized and opened for interaction with the user.

After the alpha-Editor has been opened, the alpha-Properties component takes over the central role and coordinates the interaction of the other components in response to

events within the system. Events originating from the alpha-Editor will be forwarded to the alpha-Properties component. The alpha-Properties component will then delegate a response from another component based on the rules defining the business logic. Any changes to the adornment model are delegated to the alpha-Adaptive component. If an α-Card is altered, the task of persisting the new version is delegated to the alpha-VerVarStore component. When appropriate, these new versions may be propagated to the other peer α-Docs via the alpha-Overnet and alpha-OffSynch components. The alpha-Overnet provides a generalized means for transferring data between α-Cards and the alpha-OffSynch provides the means to execute off-line synchronization.

4.4 Off-Line Synchronization

The off-line synchronization capability provided by the alpha-OffSynch component provides greater operational flexibility for the α-Flow system, but also increase the complexity of managing an α-Card's evolution. The most critical concern is the ability to correctly order the changes made.

4.4.1 Detecting Synchronization Anomalies

Two anomalies, out of order arrival and concurrent changes, are possible and must be dealt with. Figure 4.4 depicts these two anomalies. This example depicts two participants, A and B, which are exchanging updates to the same α-Card. The first section depicts the out-of-order arrival and the second section depicts concurrent changes.

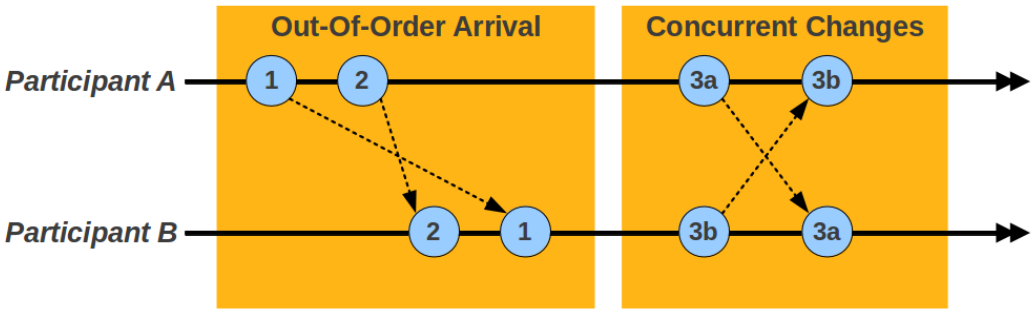


Figure 4.4: Synchronization Anomalies

α-Flow employs a versioning vector to determine the logical order of changes that may be made concurrently by various participants or arrive out-of-order. As described in [Wah11], a versioning vector consists of a set of numbers, one associated with each participant. The numbers represent the number of changes that have been made by each participant. For example, given three participants, A, B and C. If participant A has made three changes, participant B has made two changes and participant C has made one change, then the logical versioning vector would be equal to {A=3, B=2, C=1}.

The described version vector is depicted in Figure 4.5. It should be noted that these version vectors are only employed for the synchronization purposes and do not directly correspond to any version persisted within the alpha-VVS subsystem.

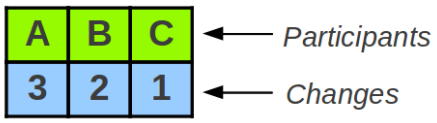


Figure 4.5: Example Version Vector

Through including these versioning vectors into the exchanged message, the out-of-order arrival and concurrent changes may be detected by comparing the incoming versioning vector with the most recent one maintained. Versions are the same if all elements are equal. A version is less than another if all elements are less than or equal to the compared version vector. A version experiences concurrent changes when one element is less than its respective counterpart and another is greater. Figure 4.6 depicts the associated versioning vectors when applied to the previously described synchronization example.

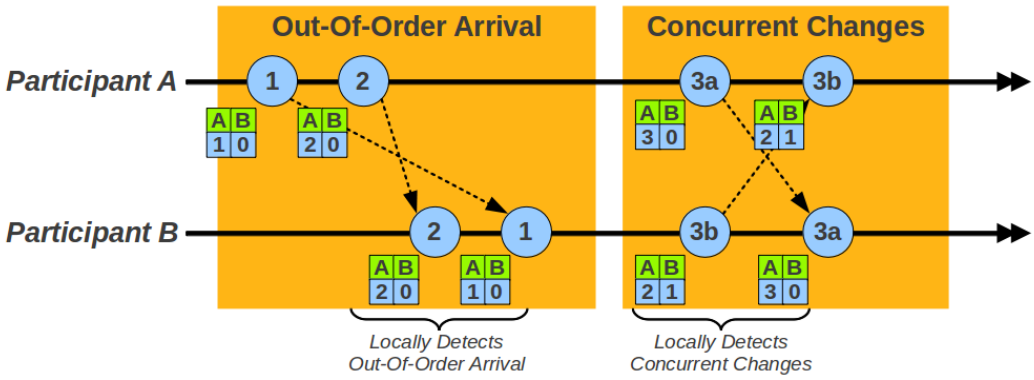


Figure 4.6: Version Vector Detection of Anomalies

4.4.2 Reconciling Detected Anomalies

Dealing with an out-of-order arrival simply requires the insertion of the incoming version before the appropriate version or reordering the versions. These two options are logical equivalent. Dealing with concurrent changes represents a serious problem, because the determination of the desired result of concurrent changes often requires human interpretation and intervention. However, human intervention is not a desired strategy in this system. The system must detect and respond to the situation appropriately. The system's reconciliation strategy results in the combining of the versioning vectors and the creation of a new combined version according to the rules defined within the alpha-Properties business logic. For example, the reconciliation of the current changes in the

previous example would result in a versioning vector of {A=3, B=1}. This reconciliation process is depicted in Figure 4.7. It is important to notice that both participants have locally arrived at the same coherent state, i.e. have the same versioning vector.

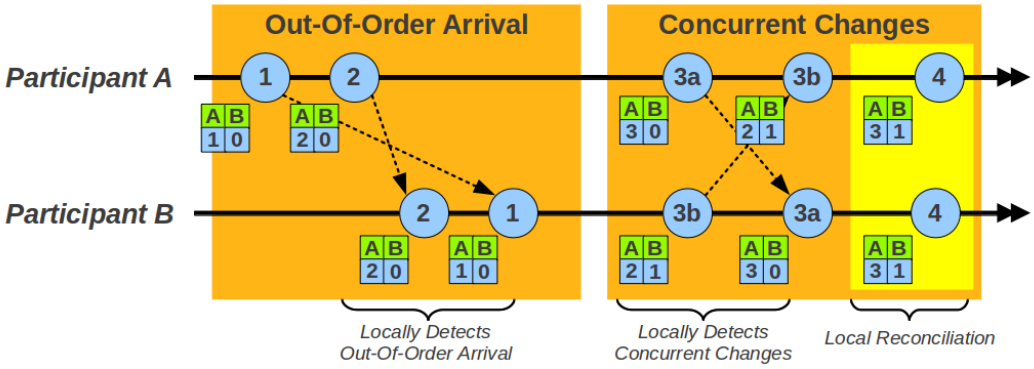


Figure 4.7: Local Reconciliation of Concurrent Changes

The automatic derivation of the appropriate content to be represented in the new version is not possible. It cannot be accomplished programmatically without making questionable assumptions about the intent of the participants. Additionally, since decisions and thus actions within this workflow are made based on the given set of information, it cannot even be assumed that a participant would make the same change if given prior knowledge to the other's change. Therefore, the only conceivable action that can be automatically executed is to set the new versions content to the first previous non-conflicting content. For example, the content of version *4* in the previous case would be equal to the content in version *2*. Figure 4.8 depicts the resulting valid path for this case. As previously described, the validity of versions *3a* and *3b* cannot be assumed, but they must be maintained for auditing purposes. Thus the valid path leads from version *4* to version *2*, skipping both questionable versions.

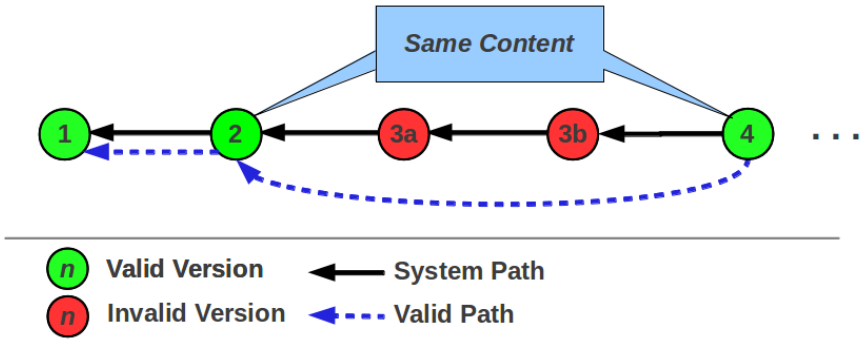


Figure 4.8: History Depiction

4.5 Alpha-VerVarStore

The alpha-VerVarStore is the α-Flow component responsible for supporting the system's versioning needs. In this section the current implementation, which was developed with the intention to be replaced through this work, and its shortcomings will be described.

4.5.1 Current Implementation

The minimalistic versioning needs of the system is to store every change made to an α-Card locally and make them available as needed. This goal defines the core of every VCS and can be implemented in numerous ways. The current alpha-VerVarStore implementation employs one simplistic method. It stores each version within a folder hierarchy in the file system and manages the versions as a map between the α-Card identification and the payload. Each α-Doc has a single root versioning folder which bears the name of the α-Episode. Within the root folder, each α-Card maintains a single folder named according to the α-Card's unique identification. Each α-Card folder contains a sequence of serially numbered folders. Each of these numbered folders represents a version of the designated α-Card and contains the respective version of the α-Card's descriptor and payload. This versioning persistence structure is depicted in Figure 4.9.

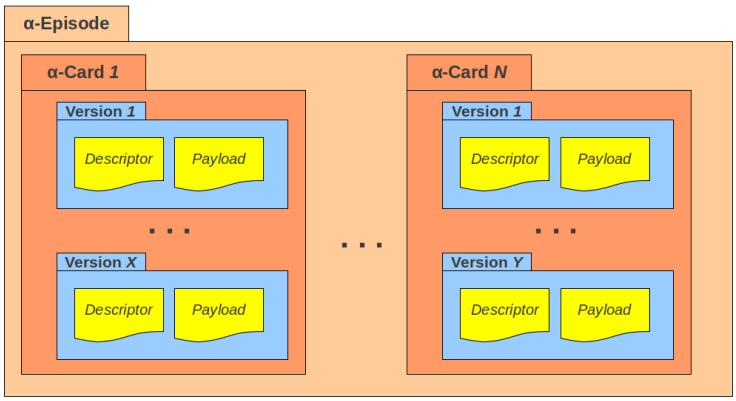


Figure 4.9: alpha-VerVarStore Persistence Structure

4.5.2 Shortcomings

This implementation is extremely lightweight with respect to source code size, platform independent and simple to understand. However, there are a number of shortcomings:

- No Version Metadata Maintained. It does not record the party responsible for generating this version, the reason it was created or the time it was created.
- No Versioning Abstractions. There is no concept of a version within the system. Additionally, it lacks the ability to abstractly consider the evolution of an α-Card based on the relationships between versions and their predecessors or successors.

- Unable to Restore a Coherent State of the α-Doc. The interdependencies between α-Card versions are not maintained. The only coherent state that can be guaranteed to be restored is the most recent version created.
- Requires Complete Storage for Each Version. While the implementation can be accomplished with negligible source code, the resulting required space needed to store each version is not efficiently used. This is because each version stores its own exclusive copy of the artifacts. It does not take advantage of any compression techniques, sharing of persisted artifacts or differentials.
- Requires Low-Level File Manipulation. The simplistic implementation does not take advantage of any versioning abstractions and thus must explicitly manipulate and manage each version through low-level file access mechanisms.
- No Means for Querying or Manipulating Histories. It is only capable of storing a version and retrieving a version. It does not provide the means to query versions or alter a previously persisted version as is needed by the system as described in Section 3.2.5: *Support Global Conflict Resolution Schemes* on page 10.
- No Distinction of Version Validity or Valid Path Depiction. There is no means for maintaining the determined validity of a version. Likewise, it is thus impossible to describe the valid evolution of an α-Card.

As alluded to in the introduction, the goal of this project is to replace the current alpha-VerVarStore implementation with a more capable implementation fulfilling the versioning needs of α-Flow.

4.6 Summary

In this chapter, the α-Flow project was analyzed. First, the key concepts, case handling paradigm and document-centered collaboration, upon which the project is built were introduced. Next, α-Flow's domain model and architecture were described. Additionally, the off-line synchronization anomalies experienced by α-Flow were described and the applied solution of versioning vectors was introduced. Finally, the current versioning implementation alpha-VerVarStore supporting the project was introduced and its shortcomings were identified.

5 Analysis – Version Control Systems and Their Evolution

Version Control Systems (VCSs) play an important role in software development and other fields where the recording of changes to electronic documents, artifacts, is either required by law or otherwise advantageous. As software developers are the primary target audience for these systems, much of this chapter will focus on the support for the demands of software development. Since its introduction, VCS technology has continued to rapidly evolve to suit the changing needs of its users. However, the basic functionality of VCSs remains invariant. This functionality can be summarized as three capabilities:

- Record changes made to designated artifacts.
- Allow artifacts to be reverted to a previously persisted version or state.
- Maintain a record of the parties responsible for introducing each artifact change.

Today, VCSs usage and their expected capabilities have grown. The following is a listing of other key features are assumed to be provided as well:

- Support for concurrent development through branching.
- Support for merging of concurrent changes into a single coherent resulting artifact reflecting the sum of the changes made.
- Support for distributed and collaborative development of any number of developers.
- Integration into software development and build environments.

In this chapter we will survey the various key influential VCSs and their position in the overall VCS technology's evolution. After a comparison of these VCSs and a conclusion will provide a short summary of the findings.

This chapter will start at the origins of VCS technology with the Source Code Control System (SCCS), progress through the client-server oriented VCSs to the most recent peer-to-peer architectures representing the latest evolutionary trend. Each VCS will be introduced and their key influences will be covered.

5.1 Source Code Control System

The Source Code Control System (SCCS) is generally accepted as the first VCS and was introduced in the early 1970s by Marc J. Rochkind, as a member of the Technical

Staff at Bell Laboratories. The abstract of his Institute of Electrical and Electronics Engineers (IEEE) publication over the SCCS set the foundation upon which all future VCSs would be built.

> *The Source Code Control System (SCCS) is a software system tool designed to help programming projects control changes to source code. It provides facilities for storing, updating, and retrieving all versions of modules, for controlling updating privileges, for identifying load modules by version number, and for recording who made each software change, when and where it was made, and why.* [Roc75]

5.1.1 Deltas

In his paper, Rochkind introduced the term ***delta*** to refer to the discrete set of changes made to an artifact in a single editor session and describes the series of deltas as a chain that may be used to describe the discrete states or versions of the artifact, which he terms levels. He also lays the groundwork and motivation for the need to support tagging and concurrent variants, which he consolidates under the term release. All deltas are consolidated in a single file but are organized according to the release for which they are applicable, allowing a stable release version to be provided for testing while continuing development on the next release. This concept is illustrated in Figure 5.1 where the triangles represent the deltas and the numbers underneath represent the release for which the delta was produced and the level or order that the delta was created. In this approach, referred to as ***forward deltas***, the original version is maintained as a full copy and any changes are maintained as a deltas that may be sequentially applied to the original version.

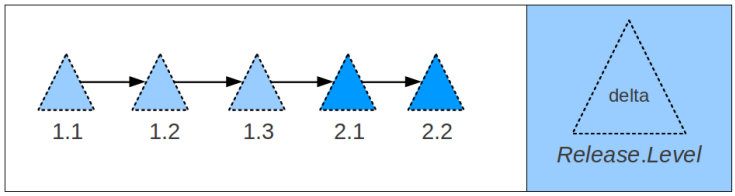


Figure 5.1: SCCS Delta Visualization

The system was designed that if another delta was introduced in a previous release, it would not be applied in the next release. Therefore, in the previous example if another delta, *delta 1.4*, was generated for first release, then it would not be applied in the second release even though *delta 1.1* through *delta 1.3* would be applied before *delta 2.1*.

Additionally, two special deltas were defined: (1) optional and (2) including/excluding. The optional delta that would only be applied when specifically requested and the including/excluding deltas were used to indicate inclusion or exclusion of other deltas. These deltas were primarily used to specialize the product for a specific customer or to

ignore a delta that was found to have introduced a fault. Using these special deltas the resulting source code could be configured beyond the linear structure described by the basic deltas.

5.1.2 Other Important Concepts

This system was based on the mainframe communication paradigm, common in that era of computing, and was implemented for the IBM System/370[1] under the OS[2] and the PDP-11[3] under UNIX[4]. In order to control access to an artifact by the collaborating software developers, a pessimistic concurrency control policy was enforced by SCCS. In order to change an artifact a user must first lock the artifact, make the changes and then release the lock to allow others access to the artifact. This approach proactively prevents the introduction of contradictory changes [SS05], but also severely limits the amount of concurrent development that may occur. The resulting architecture and associated workflow, which is shared by its successor the Revision Control System (RCS), described in the next section, is visually depicted in Figure 5.2.

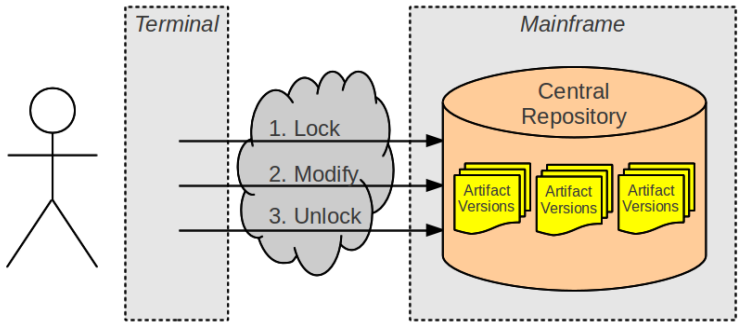


Figure 5.2: SCCS and RCS Architecture and Workflow

At the organizational level the primary unit of versioning or versioning granularity is the module, which is defined as *a convenient unit of source code, usually a subroutine or macro*[Roc75]. The module is a very fine-grained level of versioning but lacks the ability to present a unified snapshot of a system or satisfy the inter-module logical dependencies. This makes it difficult to reset the project to a specific developmental state.

In the analysis of the system's use, it was recognized that each module or artifact had on an average five deltas and about 40 percent had only one delta.

[1]International Business Machine System/370 – A mainframe computer by IBM introduced in 1970
[2]Operating System - An IBM mainframe computer operating system
[3]Programmed Data Processor - A series of 16-bit minicomputers sold from 1970 into the 1990s
[4]UNIX - A multitasking, multi-user computer operating system introduced in 1969 by AT&T

5.2 Revision Control System

The Revision Control System (RCS) was introduced in the early 1980s by Walter F. Tichy while at the University of Purdue [Tic85]. This system was based on the foundation built by the SCCS but also introduce a number of new concepts, most importantly: reverse deltas, branching and configuration management. Tichy's publication in the Software Practice and Experience journal [Tic85] provides a interesting insight into the evolution of VCS technology during this period and recognized the potential for their application outside of the field of software development.

5.2.1 Reverse Deltas

The basic versioning architecture and workflow of RCS remained largely similar to SCCS. However, the system used ***reverse deltas*** instead of the forward delta employed in the SCCS to reduce the delay experienced when checking out the most recently committed revision. In this technique, the most recent committed version is stored completely and each previous version is regenerated by applying the sequence of reverse deltas describing the evolutionary path backward to the desired version. This increased the speed of checking out the most recent version, but created a delay for checking out previous versions that increased linear with the cumulative size of the deltas that must be applied to recreate each revision. The difference in the two delta techniques is depicted in Figure 5.3.

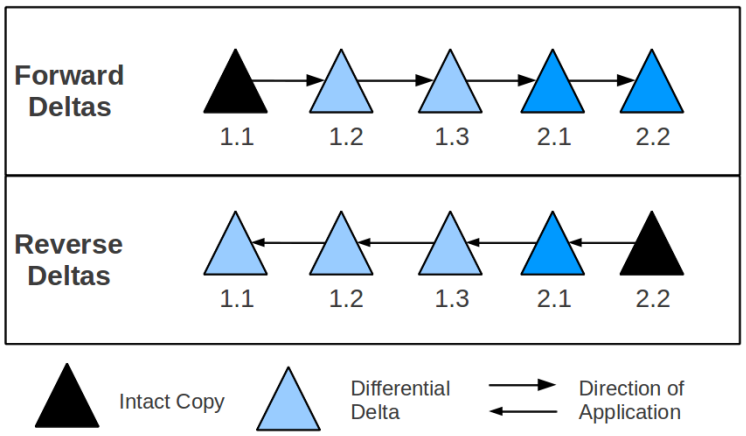


Figure 5.3: Forward vs. Reverse Delta Visualization

This delta directional change was effective, and is typically employed by all modern delta employing VCSs, because the overwhelming majority of versions accessed in a VCS are the most recent revision. However, based on typical usage profile it is the technique of choice.

5.2.2 Branching

The most profound impact that RCS had on the evolution was the introduction and argumentation for the needs of ***branches*** to support concurrent development. It was argued that branches are needed to support temporary fixes, distributed development and customer modifications, parallel development and conflicting updates [Tic85]. Whereas the temporal evolution of an artifact is depicted by its versions, a branch is used to depict an artifacts separate alternate variants. Variant is defined as "*a form ... that varies from other forms of the same thing or from a standard*" [Pea02]. The difference between a version and variant is depicted in Figure 5.4.

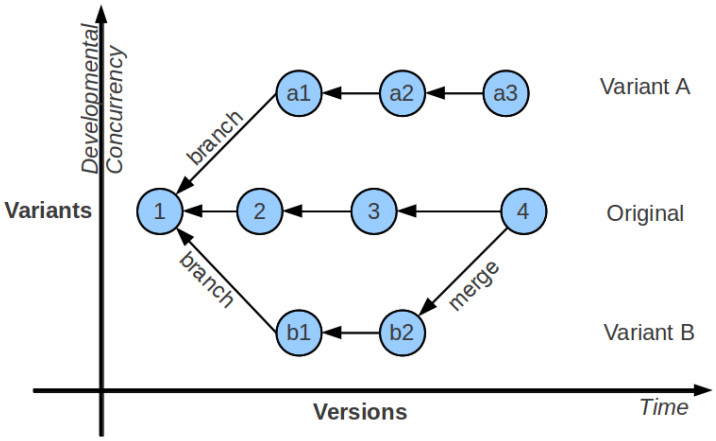


Figure 5.4: Version and Variant Difference

Each variant describes a separate evolutionary path of an artifact and therefore the number of current revisions is equal to the number of variants. However, as development progresses it is often desired that the work done on one variant be realized in another variant or branch. In order to accomplish this task, four steps must be accomplished:

1. Find a common ancestor.
2. Calculate the differences or deltas between each version desired to be merged and the common ancestor.
3. Combine both deltas and apply them to the common ancestor.
4. Create a new version with the results.

RCS supports the automation of this process through the rcsmerge command.

5.2.3 Other Important Concepts

There were a number of other interesting considerations contemplated during this timeframe of VCS technology's evolution that are reflected in Tichy's publication.

First, the system employs an extended set of attributes associated with each version to describe its status. It initially starts as 'experimental' but then may be promoted to an elevated status describing its acceptance, the examples of 'stable' and 'released' were given as such a progression. This supports the concept that each version may find itself in a differing state of validity. However, the status properties must be explicitly defined and manually updated by a user or administrator.

Secondly, the pessimistic concurrency control of locking employed was labeled as controversial. Though RCS employed the locking technique to control consistency, it recognizes the need for better concurrency support. Additionally, it allowed the restrictive locking mechanism to be deactivated if it was desired.

Thirdly, files were no longer directly manipulated within the repository. They were checked-out into a local working copy before manipulation. This is in response to the increase in the computing power available to each individual developer and the move away from mainframe computers. Additionally, multiple developers may share a single instance of a repository through the use of symbolic links while manipulating their own private working copies of the artifacts, this lays the foundation for supporting the more advanced forms of distributed collaboration associated with this system's successors.

Fourthly, RCS support is integrated into a build tool, Make [Fel79]. This indicates a trend of users demanding better integration between the tools that they employ on a daily basis. One important characteristic of VCSs today is their integration into the commonly employed Integrated Development Environments (IDEs) of application programming.

Finally, Configuration Management (CM) was discussed. While RCS employs versioning at the fine-grained artifact level, it recognized the need to consider the intricate interdependencies intrinsic in software development. A change to one file must be reflected in all associated files. RCS employs various selection criteria to allow the user to return the system to an overall coherent state of interrelated artifacts that may be compiled. The most interesting of these selection criteria are date-based and name-based selection. Date-base selection returns each of the artifacts to the state reflecting their state at the specified date in time. The name-based selection is the forerunner of tags. It accomplishes its goals by assigning symbolic names to revisions and branches. However, the symbolic names did not necessarily refer to all artifacts in the system; rather they could be assigned to any set of artifacts. Therefore, they could be used to refer to different versions of different subsystems.

5.3 Concurrent Versions System

The Concurrent Versions System (CVS) was introduced by Dick Grune in 1986 while at the Vrije University in Amsterdam, Netherlands [Gru86]. The system was developed originally as a set of 25 UNIX shell scripts as access routines for RCS, but later was reimplemented as a system of its own. It introduced two important concepts, optimistic concurrency control and the treatment of a coherent set of artifacts as a single unit.

Additionally, it extended the concept of a client-server architecture that was in the meantime supported by the RCS.

5.3.1 Optimistic Concurrency Control

It was realized that optimistically controlled concurrent development could be accomplished by the employment of two subsystems: (1) a VCS and (2) a program capable of detecting conflicting changes and merging the differences between two artifacts. In order to support concurrent operations the definition of acceptable results must be defined. A conflict between two concurrent changes can be simply described as when each concurrent change attempts to change the same line of the same artifact.

Optimistic Concurrency Example

A more thorough and thought provoking example is provided in Grune's publication [Gru86]. Assume two developers, each with their own private copy version P of an artifact obtained from a shared repository. Each developer then makes their own changes to their private copy. The changes, Δ_1, made by the first developer results in the creation of a new version Q while the changes, Δ_2, made by the second participant results in the creation of a new version R. It should be noted at this time versions Q and R represent variants of the same artifact. The two sets of changes may each be considered as transforming functions acting on P and may be mathematically defined as:

$$Q = \Delta_1(P) \text{ and } R = \Delta_2(P) \text{ [Gru86]} \tag{5.1}$$

The application of deltas is not a communicative operation. Therefore, both results of their sequential application must be considered.

$$S_{12} = \Delta_2(\Delta_1(P)) \text{ and } S_{21} = \Delta_1(\Delta_2(P)) \text{ [Gru86]} \tag{5.2}$$

If the result of both sequential applications produce the same result (i.e. $S_{12} = S_{21}$), then there is no conflict and the optimistic concurrency is successful and may be automatically merged by a program. Otherwise, if they do not produce the same result (i.e. $S_{12} \neq S_{21}$), optimistic concurrency fails because of the lack of symmetry within the operations and the true resulting version must be manually derived. Figure 5.5 on the following page provides a visual representation of this concept where the initial state, a successful scenario and unsuccessful scenario are depicted by the three separate graphs.

As a result of this consideration, each artifact could be altered in a truly concurrent manner. Each collaborating participant could concurrently and free from the actions of other participants alter the set of artifacts in their private working copies that they had checked out without having to explicitly locking the artifact prior to editing it.

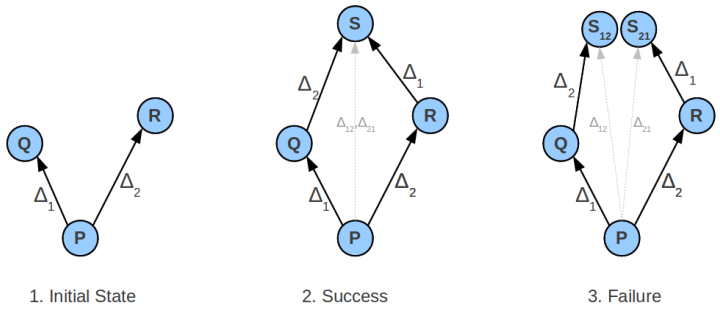


Figure 5.5: Optimistic Concurrency [Gru86]

This changed the collaboration workflow from the pessimistic *lock-modify-unlock* to an optimistic *copy-modify-merge*, facilitating elevated support for concurrent operations.

5.3.2 Client-Server Architecture

In order to support distributed collaboration, CVS extended the capability of RCS to operate on a remote repository and the concept of private working copies into a client-server architecture. This allowed a single repository server to support an arbitrary number of distributed developers collaborating concurrently. The unification of these concepts brought about the next generation of VCS technology. The previously described optimistic workflow and architecture employed by CVS and its successor, SVN, to be described in the next section, is depicted in Figure 5.6.

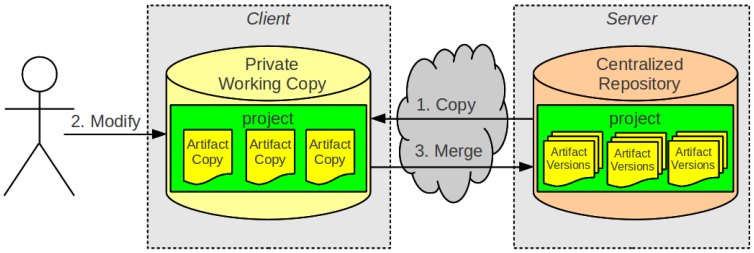


Figure 5.6: CVS and SVN Architecture and Workflow

5.3.3 Project Versioning Granularity

CVS introduced a new higher level of granularity to versioning. It allowed a coherent set of artifacts be versioned as a single unit and allows the controlling system to maintain and recreate versions of the coherent set instead of requiring the specific identification of each separate artifact's desired version which may or may not reflect an overall coherent state. Versioning at the set level takes into account the inherent interdependencies between the various artifacts of the set.

A project in software development may be considered a set of source code artifacts exhibiting intricate interdependencies and coupling. A change to one artifact must be reflecting in all related artifacts, or the overall state becomes incoherent and may not be compiled into an executable program. Through the introduction of coherent sets as a versioning granularity, the developer was able to commit the current state of the project and revert to any previous state. Previously, a developer had to attempt to recreate a coherent set through the specification of numerous different versioning numbers, one for each artifact within the project.

This concept also improved CM by bringing the versioning concept to another level where an entire project could be considered a coherent set and coherent versions of project could be easily reproduced. Prior systems required that the correct version of each artifact be specifically identified in order to reproduce a coherent state of the project that could be compiled.

5.4 Subversion

Subversion (SVN) was introduced in 2001 by CollabNet, Inc. as the successor of CVS [PCSF08]. However, it was never intended to introduce any new concepts or groundbreaking work. This can be best deduced from the authors' admission in the reference manual, or red book, preface.

> *Subversion was designed to be a successor to CVS, and its originators set out to win the hearts of VCS users in two ways by creating an open source system with a design ... similar to CVS, and by attempting to avoid most of CVS's noticeable flaws. While the result isn't necessarily the next great evolution in version control design, Subversion is very powerful, very usable, and very flexible.* [PCSF08]

While not groundbreaking, it did introduce the concept of atomic commits. This ensures that the coherent set of artifacts was persisted completely or not at all, avoiding the introduction of incomplete states into the VCS. Additionally, it introduced a common version number that is shared by all artifacts instead of each artifact maintaining its own version count.

5.5 Towards Distributed Version Control

The client-server architecture adequately supported the collaborative developmental needs for the majority of industrial or company-centric software. However, this centralized approach has a number of limitations that challenge its employment in the open source or other loosely affiliated developmental environments. A listing of these shortcomings [Muk05] of the centralized server based versioning may be summarized as:

- Single point of failure. If the single server fails, the entire developmental progress is hindered and development may only continue when the server's capabilities are reinstated.
- Single point of vulnerability. If an unauthorized intruder gains access to the centralized server, they may compromise the server's services and the integrity of the stored data. Once again hindering the overall developmental progress.
- Exclusive and Singular Ownership. The centralized paradigm endows exclusive ownership and control over the developmental progress to a single controlling party for the entire project. This restricts the capability to subcontract and outsource development portions of the project to other responsible parties.
- Fixed location. The server resides at a single fixed physical location. If the client's connection to the centralized server fails, then support for their collaborating effort is lost.
- Does not scale with needs. If the number of collaborating individuals exceeds the expected load, then the server may run out of resources necessary to support the needs of all contributors. Once again limiting the overall developmental progress. Additionally, since the majority of the computation takes place on the server, the workload is not optimally distributed across all available resources. Leaving the client machines underemployed and the server overloaded.
- Additional maintenance costs. The client and the server must maintain appropriate configurations specifically designed to support the client-server architecture. Updates made to the server must simultaneously be realized on each of the clients in order to maintain interoperability.
- Interoperability of Heterogeneous Systems. Similarly the client-server architecture is tightly coupled to the supporting architecture. This does not allow individual clients to configure their systems to match their needs. Rather, they must adhere to the strict requirements needed to provide the necessary interconnectivity.

In the mid 2000s a number of new VCSs, such as Monotone[1] [HO11], Mercurial[2] [O'S09] and Git[3] [Loe09], were introduced employing a peer-to-peer repository paradigm in attempt to overcome these limitations. Much of this effort was driven by the open source environment as it sought to ease the integration of an arbitrary number of distributed contributors operating independently and employing heterogeneous and often conflicting environments.

While each of the systems provides its own variant of support, all share the same concept of peer-to-peer repositories in order to improve the availability and support the loose collaboration of individuals that may join and leave the effort freely. In this paradigm each peer repository maintains its own redundant copy of the data and pro-

[1]Monotone Homepage: `http://monotone.ca`
[2]Mercurial Homepage: `http://mercurial.selenic.com`
[3]Git Homepage: `http://git-scm.com`

vides collaborative services to other peer repositories. The presence of redundant copies increases the availability and provides an implicit backup of data, conceptually best summarized by a quote attributed to Linus Torvalds, creator of Linux.

> *Only wimps use tape backup: real men just upload their important stuff on ftp, and let the rest of the world mirror it!* [Loe09]

However, the presence of replicated data introduces the challenge of maintaining consistency within the system which must be reflected in the system's workflow. This problem is not present in the centralized paradigm, because there is only one canonical correct copy that reflects the evolution of the project, which resides on the server. In the peer-to-peer paradigm each repository maintains its own perceptions of the artifact's evolution, which may differ dramatically. These divergent perceptions require an alteration to the basic versioning workflow. First, the alterations made in a peer repository must be retrieved, providing an overview of the divergent evolutions. Next, the changes made in both repositories must be merged together. Then, optionally the resulting state may be returned to the peer repository. This may be summarized as a *pull/copy-modify-merge/push* workflow. A visual depiction of the peer-to-peer architecture and workflow is provided by Figure 5.7.

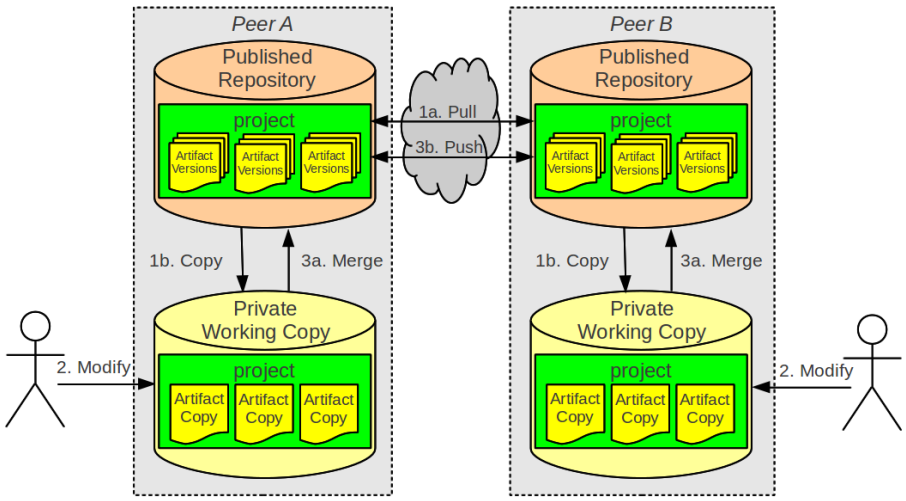


Figure 5.7: Distributed VCS Architecture and Workflow

The client-server architecture provides sequential consistency [Muk05] while the peer-to-peer architecture provides a lower level of consistency known as causal consistency [Muk05]. The client-server architecture provides sequential consistency as it maintains only a single canonical copy, which must be sequentially updated by the various clients and restricts the updates to only clients that merge their changes into the most recent version maintained by the server. However, in the peer-to-peer architecture a consistent shared state may only be achieved when all peers have shared their divergent perceptions of the artifact's evolution.

5.5.1 Collaboration Workflows

The strictly hierarchical organization present in the client-server architecture, where there is a single server which receives the updates of all clients and maintains a single *correct* canonical copy, is not inherently present in the peer-to-peer architecture. This allows the peer repositories the flexibility to assume any topology that best suits their organizational structure. However, if a clear collaboration workflow policy is not established, an uncoordinated free-for-all environment may ensue and stifle or derail the developmental progress. The goal of a collaboration workflow, is to provide the structure needed to ensure that all collaborators move forward in unison towards a common goal. In these workflows a *correct* copy representing the achieved forward progress is made available to all developers in the form of a ***blessed repository***. This blessed repository may only be updated by designated individuals after reviewing an accepting proposed changes by the individual collaborators. All collaborators then may then pull the most recent stable state from the blessed repository and propose further changes based on this common shared state [Muk05] [Loe09].

One concrete example of a collaborative workflow is the integration manager workflow [Muk05]. Within the integration collaboration workflow, all developers push their changes to public repositories. An integration manager then pulls and reviews these purposed changes from these public repositories. Based on the integration manager's testing, purposed changes may either be accepted or denied. Once a new stable state has been established, the integration manager pushes it to a blessed repository and feeds the next cycle of development. Each development cycle consists of four separate steps: update, modify, propose and accept. Figure 5.8 provides a visual depiction of the described workflow.

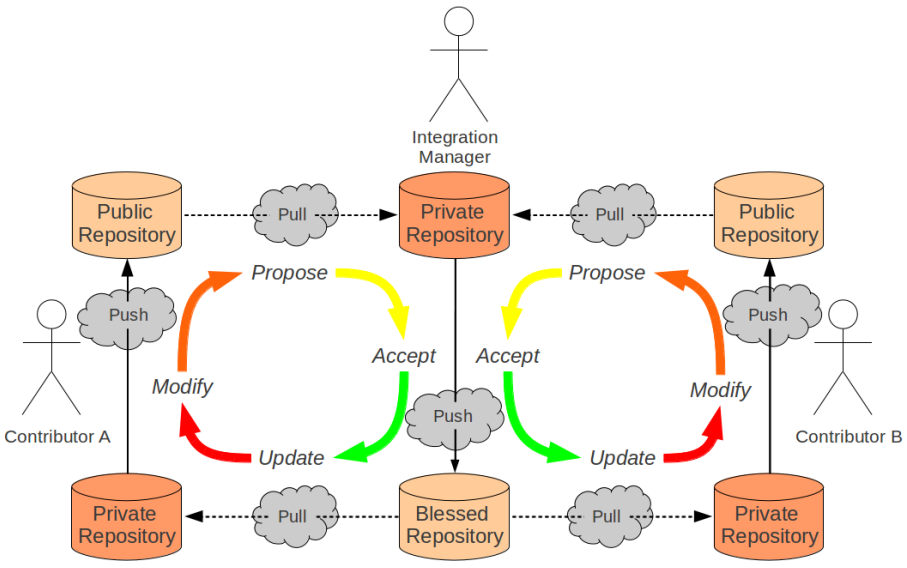


Figure 5.8: Integration Manager Collaboration Workflow

5.6 Git

Git[1] was initially introduced in 2005 by a group of Linux developers in response to the increasing restrictions placed on the free version of the commercial BitKeeper VCS, which was at the time used to support the development of the Linux kernel. Mercurial, another distributed Version Control System (dVCS), was created at the same time with the same vision provided by Linus Torvalds; contributing to the uncanny likeness of the two systems. Both drew heavily from the predecessor's BitKeeper and Monotone. However, in the end Git was chosen. Git's most sought traits include its support for distributed development, speed and maintenance of integrity and trust. The information presented in this section is derived primarily from [Cha11], [Git11], [Loe09] and [Muk05].

5.6.1 Branching Philosophy

Git's branching philosophy sets the tone for its support for distributed development and toted merging capability. Git proposes that every branch is equal and every developer's local working copy may be perceived as simply another branch by any other developer. This elevates the global conscientiousness of the overall developmental scenario beyond the local repository and facilitates the simplicity of the distributed development [Loe09].

5.6.2 Full Copy Object Storage

Much of the system's performance can be attributed to its internal storage structure. Git, unlike most other VCSs, does not use differentials to describe the different versions of an artifact. Instead it stores a complete copy of every artifact's version, generally referred to as objects. Therefore, any version of any artifact can be directly retrieved and manipulated instead of needing to be regenerated through the application of any number of differentials. The natural trade-off to this approach is the increased amount of storage space required.

However, when considering the current trends in computing this may be an acceptable trade off. Only rarely does one exceed the average computer's capacity but it is quite simple to drive a computer to its computational limitations. Given a single source code artifact averaging 16 Kilobytes (KBs), the average artifact size in the Linux kernel[2], almost 33 million versions must be created to fill a 500 Gigabyte (GB) storage capacity if no sort of compression is applied. However, Git employs a compression algorithm for its storage and may pack older versions together using delta compression to save space if needed [Loe09] [Muk05].

[1] Git Homepage: `http://git-scm.com`

[2] Linux Kernel Statics: `http://www.schoenitzer.de/lks/lks_en.html`

5.6.3 Object Integrity and Identity

To ensure data integrity, each object is identified through its 160-bit Secure Hash-1 (SHA-1) hash function. According to the National Institute of Standards and Technology, [t]*he SHA-1 is called secure because it is computationally infeasible to find a message which corresponds to a given message digest, or to find two different messages with produce the same message digest.* [Bur95] While a collision is theoretically possible, the likelihood of it happening is almost non-existent as there is about 10^{48} possible resulting hashes [Loe09]. This hash is also used to support Git's content-addressable storage paradigm, where each stored object is referenced to by and stored in a location identified by its unique SHA-1 hash value [Muk05].

Identifying objects according to their unique hash value also has the benefit that to compare the equivalence to any two objects, regardless of their size, is simply a comparison between their hash values. This also applies to the comparison of inter-repository objects, where only their hash values, not the actual object, must be exchanged to produce conclusive results. Additionally, since versions are persisted and located according to their hash values any version of any artifact that has the same content as any other previously persisted artifact version can simply reference the previously stored content and thus requires no additional space [Loe09].

5.6.4 Versioning Model

Git's internal version model is clean and straight forward. There are two kinds of elements stored: Objects and References. There are three types of Objects, each uniquely identified by their SHA-1 hash value: Blob, Tree and Commit.

The Blob is a Binary Large Object (BLOB) and roughly corresponds to any versioned artifact or file. A Tree references and maintains the names of a set of Blobs or other Trees and roughly corresponds to a directory or folder. This separates the name from the content and facilitates the reuse of Blobs for artifacts with the same content but different names.

The Commit records a snapshot of the versioned artifacts, through a reference to the parent Tree, the author, message, other metadata pertaining to the committed state and a listing of the preceding or parent Commits. Generally, in a linear developmental evolution there is only a single parent. However, merges may result in multiple parent Commits and the initial Commit has no parent Commit. Git's Octopus merge allows the user to merge any number of states. However, it is actually just the recursive application of pairwise merges, which are then smashed into a single merged state [Loe09].

References maintain a named relationship to given Commits and are commonly applied as tags or branch heads and provide alternate means of identification. The versioning model used by Git is depicted in Figure 5.9 on the facing page.

However, querying of a single artifact represents a challenge for this structure. To investigate the history of a single artifact requires the traversal of the tree structure in every commit. Likewise, the change set of a set of artifacts also requires a search of the

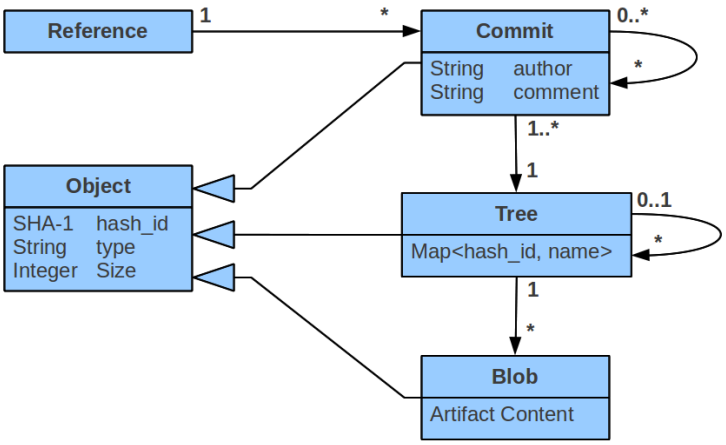


Figure 5.9: Git Versioning Model [Muk05]

respective commit tree structures.

5.6.5 Rebasing

Git provides means to seemingly alter the recorded history through its rebasing facilities. Rebasing allows a developer to delete a commit, change the order of commits and change the content of a commit or its metadata. However, the alteration of history is an illusion.

Because of its strong use of cryptography, to prevent corruption or tampering of data and the SHA-1 hash value based identification, driving its content-addressable storage paradigm, a commit cannot be changed. Rather a new set of commits must be created to reflect the desired set of changes. That is why it is strongly recommended that a history is not altered once it has been published or shared, because the appearance of the new commits, which reflect the set of changes described by the old commits, may confuse the system and lead to an attempt to remerge the differences [Loe09].

Consider the situation where a developer creates three commits A, B and C and then attempts to change the order of commits B and C through rebasing. The result is the creation of two new states C' and B', each with a different SHA-1 hash value then their original commit. In unpublished histories this generally causes no problems. However, if the history had been published, all other repositories would still consider C instead of B' the head and would lead to confusion when the repositories attempt to resynchronize, through a pull operation. Ultimately, the developers would be required to remerge all changes from the common ancestor A. This scenario is depicted in Figure 5.10 on the next page.

5.6.6 Other Important Concepts

In order to facilitate exchange in a heterogeneous developmental environment, Git also provides the ability to synchronize with repositories of other VCSs such as CVS, SVN

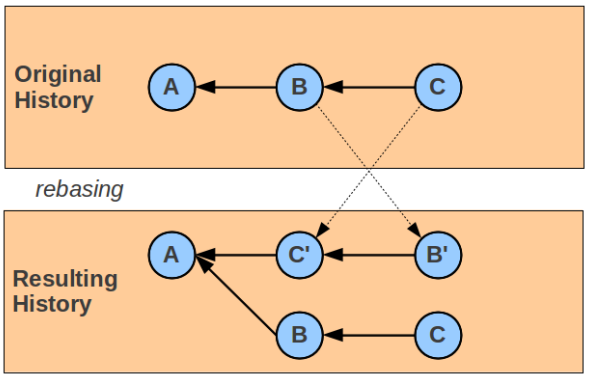


Figure 5.10: Git Rebasing

and Mercurial. It may also act as an intermediary between two other VCSs.

An overlying drawback for Git, is its platform dependency. Git is designed specifically for employment on Unix/Linux based machines, to include apple's OS X. While there is a Windows implementation, it is not the primary focus of development and limits its application on Windows based machines [Muk05] [Loe09].

5.7 Mercurial

Mercurial[1], as described before, shares a similar heritage and vision with Git. As such, the majority of the systems capabilities are comparable with Git. The greatest differences are its platform independence, versioning model and object storage structure. It is predominately implemented in Python, but also has portions implemented in C for performance reasons. This makes, especially with respect to Git, more or less platform independent. The information in this section is drawn from [O'S09] [Mer11] and [Muk05].

5.7.1 Versioning Model

Like Git, Mercurial has two basic types of elements: Tags and RevLogs. Likewise, the purpose of the two elements is similar to Git's References and Objects. The RevLog is composed of of two functional units: the Index and the Data file. The Index contains the RevLogs metadata, including SHA-1 hash value providing its unique identification, while the Data component maintains the content. The ChangeLog, Manifest and FileLog roughly correspond to Git's Commit, Tree and Blob Objects as previously describe. However, the FileLog maintains all versions of a given artifact instead of a single version. This allows the history of a single artifact to be retrieved easily. Mercurial's versioning model is depicted in Figure 5.11 on the facing page.

[1]Mercurial Homepage: `http://mercurial.selenic.com`

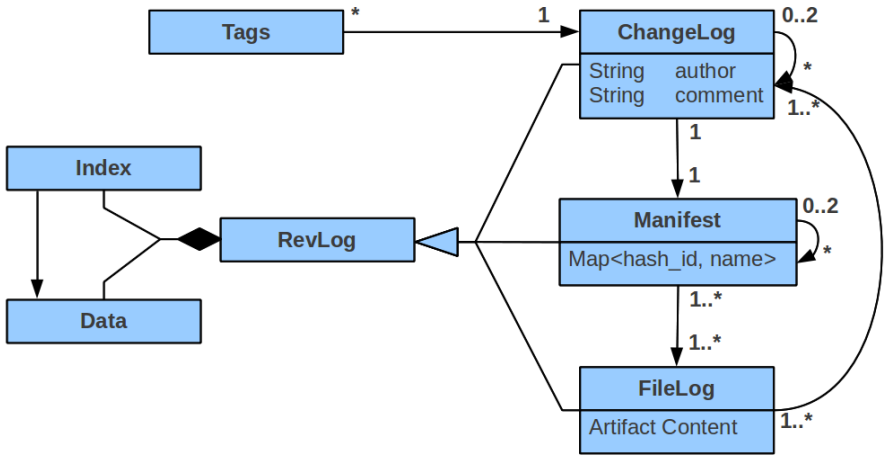


Figure 5.11: Mercurial Versioning Model [Muk05]

5.7.2 Differences from Git

The most significant difference between the two systems is that Mercurial normally stores the differential between versions and not a complete copy of each version. When the cumulative size of the deltas required to reproduce a given version is greater than the file itself, a complete version of the file is stored and the process begins again. In order to recreate a given version, the most recent complete copy before the designated version is retrieved and the remaining sequential of deltas is applied. This accounts for an efficient method of storage, producing a repository much smaller than SVN. However, Git's compression techniques result in a smaller repository[1] [Muk05].

Other differences include: (1) number of parent versions, (2) history alterations and (3) repository philosophy.

1. *Parent Versions.* A ChangeLog can only have at the most two parents, limiting its merging representation to two states. Git allows a Commit to have any number of parents, theoretically allowing any number of states to be merged.
2. *History Alteration.* A technique for altering history, similar to Git's rebasing, is not included in the basic installation. It is available, but is considered an extension.
3. *Repository Philosophy.* Finally, Mercurial works on the philosophy that all developers are working from the same repository. Whereas Git repositories are considered singular unique instances and only chosen branches are explicitly shared, Mercurial assumes all branches are shared as a global repository.

[1] Decentralized Version Control Comparison: `http://vcscompare.blogspot.com/2008/git-mercurial-bazaar-repository-size.html`

5.8 Summary

Since its origins, VCS technology has evolved to suit the needs of software development. It has evolved to support collaborative development by any number of developers distributed around the world and operating at different times. Significant support for divergent development and the reunification of these variants has been attained.

SCCS is commonly accepted as the original VCS and set the foundation on which other VCSs would build. It introduced the concept of a repository and versions. RCS introduced the concept of the variant through its branching concept. CVS unified several parallel introductions and grounded the concepts of a private working copy, the client-server architecture and versioning at the project level, which reflects the interdependencies inherent in source code. dVCSs, such as Monotone, Mercurial and Git, broke the restrictive nature of the client-server VCS paradigm by introducing the peer-to-peer architecture. Table 5.1 provides a summary of this evolution and the important concepts introduced by the key VCSs representing the evolution of VCS technology discussed in this chapter.

	SCCS	**RCS**	**CVS**	**SVN**	**Git**
Architecture	Mainframe	Mainframe	Client-Server	Client-Server	Peer-to-Peer
Storage	Forward Delta	Reverse Delta	Reverse Delta	Reverse Delta	Full Copy
Concurrency	Pessimistic	Pessimistic	Optimistic	Optimistic	Optimistic
Granularity	Artifact	Artifact	Project	Project	Project
Introduced Concepts	· Repository · Version	· Branch	· Working Copy · Project	· Atomicity	· Peer-to-Peer · Object Storage

Table 5.1: VCS Summary

6 The Hydra Approach to Versioning

This chapter defines the concepts logical units and version validity and grounds the need for a new VCS, Hydra, to support multi-headed versioning with validity tracking. Logical units are a basic elements of structural organization and will be used to organize artifacts into logically coherent subsets in support of multi-headed versioning. Once logical units and validity have been appropriately defined, techniques for employing contemporary VCSs in support of these concepts will be investigated. Finally, an assessment of available VCSs with respect to their ability to satisfy the versioning needs of α-Flow is presented.

6.1 Logical Units

In order to employ multi-headed versioning, a means of organizing the complete set of artifacts into subgroups or subsets must be first considered. The concept of a ***logical unit*** will be developed in this section to fill this structural gap. First, the concept of the logical unit will be derived from the α-Flow project and generalized for the field of software development. Next, the conceptual definition will be formalized and the benefits gained from its application will be discussed.

6.1.1 Conceptual Introduction

In this section an initial basis for discussion will be established. Consider a complete set of artifacts, P. This complete set of artifacts may be visualized as all of the electronic documents that belong to a given project. Next, these artifacts may be assigned to an arbitrary number of subsets or may belong to none of them. The subsets may be visualized as the project's subprojects. The complete set of artifacts may be broken into any number of subsets, but in this example three will be used, A, B, and C. With this basic information, an initial Venn diagram may be produced and is depicted in Figure 6.1 on the following page.

Based on a random assignment of the artifacts to the given three subsets, each artifact may find it self in one of four different situations. It may: (1) belong to exactly one subset, (2) be shared by two subsets, (3) be shared commonly by all subsets or (4) not belong to any subset. Where *exclusive* artifacts are possessed by exactly one subset, *shared* artifacts are shared by multiple but not all subsets, *common* artifacts are those belonging to all subsets and *unassigned* artifacts belong to no subset. This artifact classification, based on its quantity of possessing subsets, is depicted in Figure 6.1 on the next page.

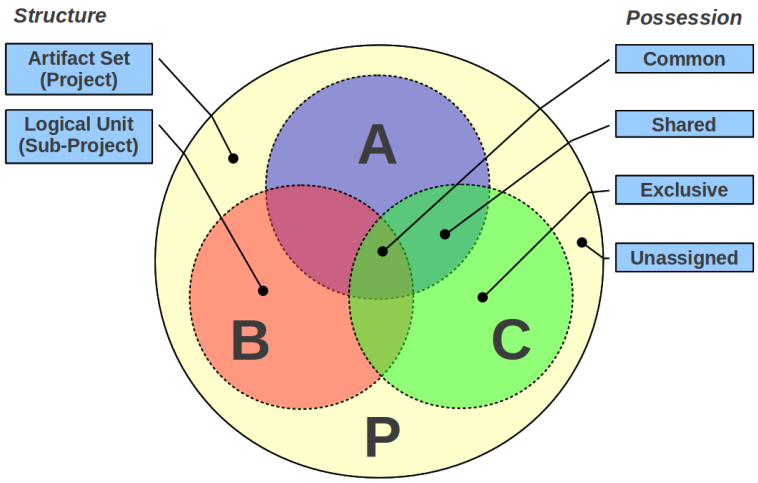


Figure 6.1: Conceptual Logical Units

In the following sections the initial concept of the logical unit will be considered with respect first to its applicability in the α-Flow project and then generally within the field of software development.

6.1.2 Logical Units in Alpha-Flow

Dealing with complexity is not the only reason for decomposing the overall project into a set of logically independent subcomponents. Some projects, such as α-Flow, present an overall structure that is naturally described as a composition of relatively independent units. Within α-Flow, an α-Doc is composed of a set of independent α-Cards.

Changes to one α-Card represent the progress of a single treatment or task within the overall distributed healthcare process and must be managed and maintained independently to allow for the autonomous actions of the medical professional. The decoupling of these specific treatment instances within the workflow provides the flexibility necessary to support α-Flow's dDPM approach and is one aspect that distinguishes it from the traditional task-oriented workflow approaches. Therefore, the logical unit within α-Flow must center around each specific treatment instance.

The core artifacts of a treatment instance are the representative α-Card's contents, i.e. the descriptor and payload. Beyond these exclusively owned artifacts, each of the coordination cards also maintain information that is important to the treatment instance. However, these coordination α-Cards are commonly shared by all content α-Cards. Considering the purpose of each coordination α-Card, it may be determined that the CRA and APA provide a single commonly shared perspective and may be considered *common* artifacts. The PSA represents the interrelationships between the various content α-Cards and may be considered to be *shared*.

Ownership or definition of the responsible party within a decomposed project is often an important concern. One party may be responsible for the project's overall success, but

often a different responsible party is responsible for managing each subproject. Within α-Flow, each α-Card has a definitive responsible party. The healthcare professional that produces the payload is responsible for ensuring that the shared information is correct. This presents the need to be able to associate ownership to the logically independent unit.

Mapping an Alpha-Doc to Logical Units

In this section an example will be presented to reinforce the concepts considered in the previous section. Consider an α-Doc consisting of the standard three coordination α-Cards and three additional content α-Cards, A, B and C. Each of the α-Cards consists of two artifacts for a total of six artifacts. These six artifacts may be organized in a number of different ways, depending on the perception of the user.

In the simplest organizational structure, each of the α-Cards may be represented as a logical unit. This results in the creation of six logical units. Another option is to encapsulate the CRA and APA into a single logical unit, which would result in five logical units.

Figure 6.2 provides a visual depiction of the simple per-α-Card scenario mapped onto the original conceptual logical unit Venn diagram. It should be noted that no artifacts are *unassigned*.

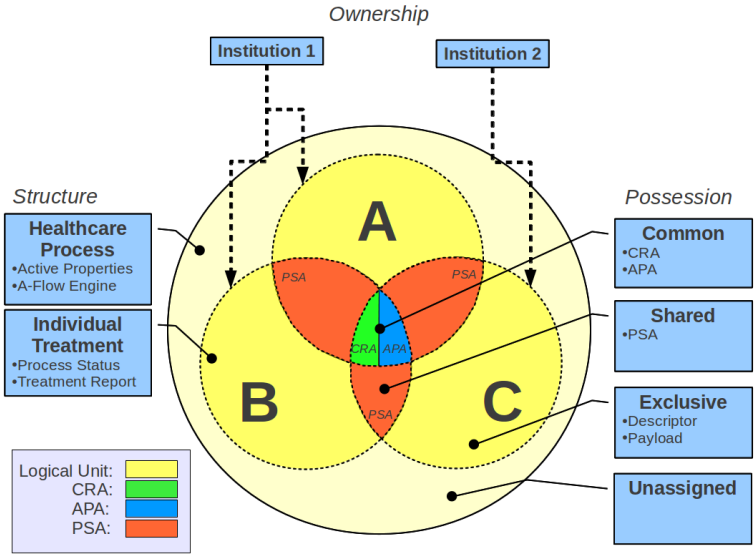


Figure 6.2: Alpha-Flow Mapping to Conceptual Logical Unit Venn Diagram

6.1.3 Logical Units in Software Development

Early software programs were composed of a single monolithic block of code capable of executing the desired task. This approach is adequate for the accomplishment of simple

tasks. However, by the 1960's and 70's software development was facing a crisis brought about by the increasing complexity of the projects that were undertaken. Projects were regularly delivered late, over budget, with residual faults and not meeting the client's expectations [Sch05].

In an effort to combat these shortcomings, software developers embraced structured programming and examined the benefits of modularity. Modularity seeks to break the project into a set of modules that exhibit low inter-module coupling and high intra-module cohesion. These properties of low coupling and high cohesion simplified development and maintenance efforts and reduce the overall cost of developing software [Sch05] [BME+07].

The theory of modularity provides a number of benefits for the development of large complex software projects. These benefits include [Sch05]:

- Breaks large complex project into independent manageable subprojects.
- Isolates each subproject from changes in other subprojects.
- Allows each subproject to be resources and developed independently.
- Simplifies maintenance and reduces effort of fixing bugs.

Based on the concepts of modularity, a large complex software project can be broken into a set of loosely coupled subprojects. However, subprojects can not be completely decoupled as they must interact to accomplish the overall project goals. Interactions between subprojects may be limited to interfaces, but these interfaces must be exposed and shared across the interacting subprojects. Additionally, some aspects, such as configuration details, must be shared and commonly available across all subprojects [Sch05].

Software Development Project Structure

At the most general level a project is composed of a set of files. Each of these files may be associated with one or more possessing elements: the project and subprojects. Technically all files belong to the overall project, but a more appropriate analysis of possession may associate them to a specific subproject.

Files normally associated with the overall project include: build environment files (e.g. *pom.xml*[1] or *makefile*[2]), user's manuals, libraries and other documentation.

Subprojects are primarily composed of source code files. Source code files may be associated with one or more subprojects. Single or exclusive possession indicates no interdependency and frees the file for autonomous manipulation of the file by the possessing subproject. Files that are possessed by multiple subprojects indicate a coupling between the involved subprojects. These shared files define the methods of interaction between subprojects and are typically realized as interfaces or facades.

[1] Project Object Model, Apache Maven construction configuration file
[2] Make utility's construction configuration file

Finally some files provide common or global knowledge or configuration details needed by all subprojects. These files are typically configuration files and changes must be appropriately reflected in all subprojects.

Mapping a Software Project to Logical Units

Consider a software project consisting of a set of artifacts P. This project is decomposed into three subprojects, A, B and C. These three subprojects may then be conceptualized as three separate logical units and it is simple to assign all artifacts that belong exclusively to an subproject to its respective logical unit.

However, the assigning of artifacts that are shared by multiple subprojects, i.e. the interfaces and configurations, cannot be as simply resolved. A shared artifact may be assign to any or all of the respective logical units. It is then up to the developer to make the most appropriate assignment.

This leaves the common and unassigned artifacts to be considered. Obviously, one or more logical unit may be created for each subset. However, there is a striking similarity between the roles that of the common and unassigned artifacts. They both belong most appropriately in the general sense to the overall project and not any subproject. This allows both of these artifact types to be encapsulated within one logical unit, which is used to represent the project in a more general sense.

The assignment of logical unit to owner is highly project dependent and should follow the same general approach in which the responsibilities for the subprojects were assigned. However, the owner of the common and unassigned artifacts should maintain an overview of the entire project, to ensure that their actions do not negatively influence any other logical unit.

Figure 6.3 on the next page depicts the situation where the project is broken into four logical units. One for each of the subproject's A, B and C and one for the project overall. The artifacts shared between A and B are assigned exclusively to either of the logical units. The artifacts shared between B and C are exclusively assigned to logical unit C. It is important to note that this is only one of many different ways in which the artifacts may be organized; their assignment is project and developer dependent.

Other Considerations

Software continuously evolves to meet the ever-changing demands of its employing environment. Customers are constantly demanding the addition of a new feature and new software is being developed with which it must interact. In order to manage the new and changing demands, the decomposition structure of the project must be constantly adjusted. Subprojects may need to be added or split when the complexity exceeds that which may managed. Likewise subprojects may need to be merged when multiple interacting subprojects are simplified to the point at which they be managed as a single unit.

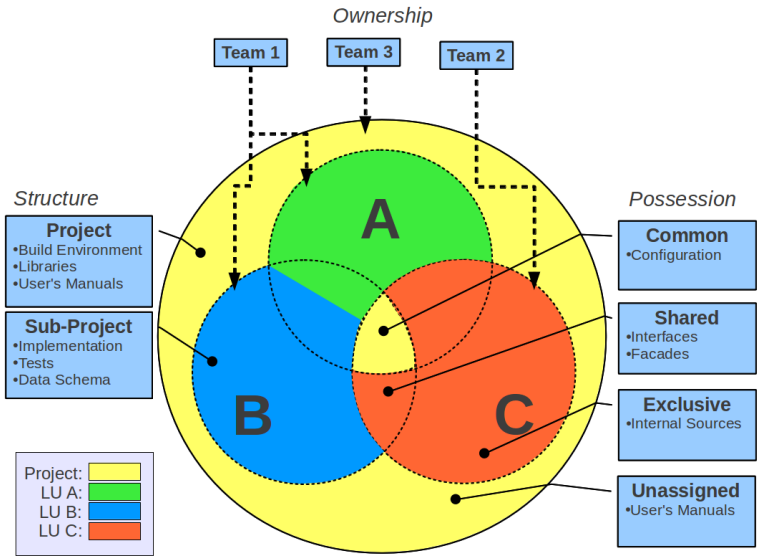


Figure 6.3: Software Project Structure and Possession

6.1.4 Definition of Logical Units

Based on the previous considerations, a logical unit may be defined as:

Logical Unit – a logically independent set of artifacts, that exhibits loose coupling to other artifacts and high intra-set artifact cohesion, which requires or benefits from an independent versioning.

Therefore, a logical unit can be described to be a subset of artifacts that belong to the project. Given a set A of artifacts a_i that make up project P and a set B of artifacts b_i that make up a Logical Unit LU_B. Then Logical Unit LU_B belongs to Project P is B a subset of A. Equation 6.1 describes this relationship mathematically.

$$P = A := \{a_0, a_1, ...a_n\} \quad : P \text{ possesses artifact } a_i \tag{6.1a}$$

$$LU_B = B := \{b_0, b_1, ...b_m\} \quad : LU_B \text{ possesses artifact } b_i \tag{6.1b}$$

$$LU_B \in P \quad : B \subseteq A \tag{6.1c}$$

The overall project may then be defined based on the set of logical units, which represent the decomposition management structure. Project P is the union of all sub-component logical units and the remaining set of artifacts Z not directly possessed by

any logical unit. Equation 6.2 describes this relationship mathematically.

$$Z := A/(\cup_{(i=0,\ldots x)} LU_i) \tag{6.2a}$$

$$P := (\cup_{(i=0,\ldots x)} LU_i) \cup Z \qquad : LU_i \in P \tag{6.2b}$$

Therefore, the logical unit is a generalized organizational structure that may be used to describe the decomposition of an overall project into subcomponent parts. And provides the context of assigning artifacts to possessing logical units.

Versioning Granularity

However, VCSs are responsible for maintaining a description of a project's progression through recording of designated versions or states. Current VCSs provide two versioning granularity levels: artifact level and project level. Neither supports an independent versioning at the subproject level.

The artifact level is fine grained versioning. It records the progression of each artifact independently, but does not account for the interdependencies inherent between coupled artifacts. The project level on the other hand, records the collective state of all the project's artifacts as a single version. This accounts for the interdependencies, but limits the representation to a single history. The logical unit may be introduced as a new versioning granularity level that provides an intermediate representation accounting for the necessary interdependencies while still supporting the recording of more than one history.

For example, given a project composed of N artifacts distributed across M subprojects. The number of histories recorded for each granularity level and whether the artifact interdependencies are maintained is depicted in Table 6.1.

	Artifact	**Logical Unit**	**Project**
Independent Histories (Qty.)	N	M	1
Interdependency Maintenance	No	Yes	Yes

Table 6.1: Granularity Levels Comparison

Project State

To complete the introduction of the logical unit as a new level of granularity, the definition of the recorded state must be considered. In order to support the intra-unit dependencies, the logical unit's state is defined as the collective state of all artifacts assigned to the logical unit. The function $S(LU)$ defines the state of logical unit LU as

the union of each of its possessed artifacts' state, $S(b_i)$, this is described in Equation 6.3.

$$S(LU) := \cup_{i=0,...m} S(b_i) \qquad : b_i \in LU \tag{6.3}$$

The state of the overall project represents the integration of the various logical units into a coherent project and may then be defined based on the states of the composing logical units. The project state maintains the inter-unit dependencies as well as the intra-unit dependencies of the artifacts possessed directly by the overall project. The function $S(P)$, described in Equation 6.4, defines the state of the overall project P as the union of each of its decomposing subprojects' state $S(LU_i)$ and the state of the set of additional artifacts $S(Z)$ possessed by the project.

$$S(P) := (\cup_{i=0,...x} S(LU_i)) \cup S(Z) \qquad : LU_i \in P \wedge Z \in P \tag{6.4}$$

Figure 6.4 depicts the distinction between the contemporary method of maintaining a single monolithic project state encompassing all subprojects and the proposed independent versioning and integration paradigm introduced by the logical unit, multi-headed versioning.

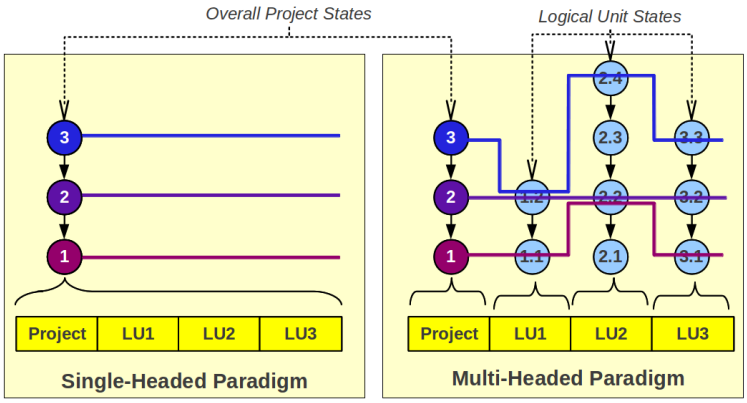


Figure 6.4: Comparison of Single-Headed and Multi-Headed Versioning Paradigms

Incoherent Project States

Through the integrated definition of the three levels of versioning granularity, a means for managing the tracking of progression of independent logical units and simultaneously the overall project is achieved. However, based on the analysis of software developmental structure and possession, an artifact may be allowed to be possessed by multiple logical units. While this should be avoided to facilitate loose coupling, the impact of its acceptance must be considered.

The possession of a single artifact by multiple logical units has the potential of creating an incoherent or undefinable overall project state. An incoherent state is produced when the shared artifact is different in at least two of the possessing logical unit's states which are used to define the overall project's state. This produces a situation where the sequence of operations that must be executed to return to a designated state fail support the commutative property. For example, given a project P with two logical units, LU_1 and LU_2, the state of the project is defined as the union of the two logical units states. However, if each logical unit state maintains a different version for a shared artifact the correct resultant version cannot be assumed. The function $R(LU_x)$ is represents an atomic action which resets the state of the artifact to a given state. The non commutative property may be described by Equation 6.5.

$$\begin{aligned} &R(LU_1) \times R(LU_2) \neq R(LU_2) \times R(LU_1) \\ &\quad : b_i \in LU_1 \wedge b_i \in LU_2 \text{ and } S_{LU1}(b_i) \neq S_{LU2}(b_i) \end{aligned} \tag{6.5}$$

6.1.5 Benefits of Logical Units

A number of benefits can be associated with the introduction of the new versioning granularity level, the logical unit. Some can be derived from the principles of project management, others from the principles of modularity and still others from their specific application in software engineering.

Logical units encourage the decomposition of a complex project into smaller, more manageable subprojects. The benefits with respect to the project management are as follows:

- Independent Management. Each of these subprojects may then be resourced, managed and developed independently while the overarching cohesive concerns, shared by the subprojects or belonging specifically to the overall project, are maintained.
- Multiple Owners. The introduction of separate subproject elements also supports the ability to assign ownership to each. Similar to assigned responsibility and delegation of duty observed in reality.
- Independent Evolution. In contemporary development, all changes are made from a single state of the overall project and all interdependencies must be satisfied. Any change to any aspect of the system creates a single overall state from which the next change must be made. This creates a restrictive lockstep approach to development, where all subprojects must proceed at the same rate. The separation of the overall project into a set of subprojects frees each from the confinement of the lockstep progress. Each project is free to advance at the its own rate, and are integrated across clearly defined and visible interfaces.

The versioning state independence of the logical units emphasize the principles of modularity. The enhanced benefits are as follows:

- Clarified Purpose and Impact of Commits. When only the project level of versioning is available, the purpose and impact of a commit is convoluted and spread across all aspects of the project. It cannot be easily determined which aspects of the project a commit is intended to affect or what the specific change impacts the overall project. However, here each commit is associated to a specific logical unit, which immediately improves the understanding of the commits purpose by defining its context at a finer granularity level.
- Explicit Representation of Interdependencies. Contemporary VCSs only maintain a single overall state of the system, which ensures all interdependencies are maintained; regardless of their appropriateness. However, this ignores the key principles of modularity and encourages shortcuts in the name of efficiency or time saving. Interfaces defining the subcomponent interaction are either not created, not used or not well designed. If not controlled, these interdependencies will increasingly tax the progress of the development. Here, interdependencies are explicitly represented and intentionally created by the sharing of artifacts across logical units. This raises the awareness of the coupling created between the logical units and forces special attention to be paid to the interfaces across which subcomponents interact.
- Emphasize Loose Coupling and High Cohesion. High coupling between logical units will force changes to one logical unit to be reflected in another logical unit, creating changes across multiple aspects of the project and contradicting the intentions of logical units. Additionally, changes to logical units with low cohesion may reflect changes to different aspects of the project; also contradicting the intentions of logical units. In order to support the independent history of the logical units, subcomponents must be organized in a manner that emphasizing their loose coupling and high cohesion.

Software engineering is different from other forms of engineering, especially considering the aspects of cause and effect. A small change to code, such as changing the sign of a integer from positive to negative, could have a dramatic effect or could have no effect at all. Additionally, development does not experience a constant monotonic increase in value, as is experienced in most constructive efforts. Occasionally faults are introduced or alternate algorithms are implemented in an attempt to find the optimal solution. The introduction of the logical unit as a versioning granularity provides the following benefits that are specific to the art of software engineering:

- Integration Optimization. In contemporary VCSs all integration done implicitly at developer level, where the priority is the creation of executable code. Minimal testing is done to see how changes to one aspect of the overall project affect the rest of the project. In general, the integrated product is found to be acceptable it it compiles and passes all the unit tests. Little attention is given to the overall optimal performance of the product. By separating the subprojects history

from the overall project's history, special attention may be made integrating the subprojects and determining the impacts of each subproject's change.

- Fault Locating. Since each commit is associated with a specific area and provides a clarified purpose, the detection of the exact location of a fault is simplified. If the fault is assumed to be within a specific logical unit, then only the commits within that logical unit must be considered. Additionally, since the purpose is not convoluted across the entire project, it may more accurately reflect the intent of the change or define what was actually changed in a commit. Also providing more information for the detection of the specific commit that introduced a fault.

6.2 Version Validity

In this section, the concept of version validity will be discussed. First, an initial preview of the version validity and valid path concepts will be introduced. Next, they will be further clarified with respect to their applicability in the α-Flow project and software development in general. Finally, a formalized definition and their benefits will be provided.

6.2.1 Conceptual Introduction

Validity is a characteristic of a version, which must be specified by the employing system. As will be presented in the following sections, validity has a different definition in the context of α-Flow as in software development. However, the commonality indicates its *acceptability* or *correctness* within the employing system. A working definition of validity is as follows:

Valid A property expressing the application-defined *correctness* or *acceptability* of a version.

Being able to differentiate between versions based on their validity allows for the separation of versions into two categories: (1) valid and (2) invalid. Observing either of these groups independently presents a unique perspective of an artifact's history that is different from the combined perspective, where all versions are included.

System Path and Valid Path

Contemporary VCSs, which are unable to differentiate between versions based on their validity, may only present the combined perspective. This perspective will be known as the ***system path*** and describes the complete history of an artifact including all valid and invalid versions.

A perspective that restricts its reflection to only valid versions, provides a depiction of the versions which are found to be acceptable within the system. This sequence of acceptable versions presents a strictly monotonic view of an artifact's history with

respect to its progression towards a *complete* or *finished* state. This progression will be referred to as the ***forward progression*** of a history and the perspective view of this progression will be known as ***valid path***. Any version that is found to be invalid is not present in this perspective. This ensures that all progress is based on a stable and acceptable version basis.

Path Example

Figure 6.5 presents a visual depiction of the system and valid paths. In this example there are four versions. Versions *1*, *2* and *4* are valid and version *3* is invalid. The system path is ignorant of the versions' validity and traces the history expected to be observed within a contemporary VCS. The valid path is *aware* of each version's validity and traces the history but only includes valid versions while ignoring the invalid version, version *3*. The cause of version *3*'s invalidity is left undefined at this point as it can only be clarified within the context of a specific application. This will be covered in the following sections.

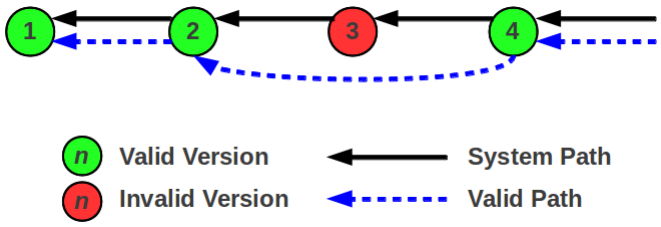


Figure 6.5: System and Valid Path Initial Concept

6.2.2 Validity in Alpha-Flow

Within α-Flow a version's validity may be affected either by the content or by the concurrent manipulation possible within the system's distributed architecture. Content-based validity typically results from some type of human error and requires a human interpretation of the *correctness* of the information present. α-Flow's synchronization algorithms are responsible for automatically detecting and resolving conflicting concurrent changes, which may result in the invalidation of previous versions.

Content-Based Invalidation

A number of reasons may cause a version to be interpreted as invalid based on the represented information. Some of these reasons are treatment specific and others may be generalized to the any paper-based process. The following is a list of some common reasons that a version within α-Flow may be invalidated:

- Clerical Error. Every time information is entered, there is the possibility for the introduction of unintentional errors. One common form of clerical error, results when a secretary inputs incorrect information in a form. Misspelled names, incorrect addresses, or otherwise incorrect personal information are common manifestations of these errors. Another may be the accidental exchange of patient information. This may occur when the diagnosis of one patient is associated with another patient's personal information.
- Invalid Test Result. Many common medical examinations include tests that are sensitive to external influences, difficult to properly execute, or may executed under false pretensions. One example of a test sensitive to external influences is a blood test. The result of a blood test may be heavily influenced by diet, age and a number of other factors. If the analysis of a blood test assumes that a patient had not eaten within the last 24 hours, the result may be skewed if the patient did not adhere to the medical professional's instructions. Another example of this type of error may occur when a doctor requests an x-ray of a patient's right hand and somehow the request is incorrectly perceived and an x-ray of the patient's left hand is produced.
- Diagnosis/Report Based on Invalidated Assumptions. Often decisions or diagnosis' are based on presumably correct information. However, as described in the previous points. Information may be invalidated for a number of reasons. Therefore, the transitive impact of invalidating one document must be considered. Any other conclusions drawn based on that invalidated document, may possibly also be invalidated.

Synchronization-Based Invalidation

Due to α-Flow's loose distributed architecture, concurrent changes may occur and must be resolved locally. α-Flow's synchronization process computationally detects conflicting concurrent changes based on a logical timestamp as described in [Wah11] and in Section 4.4: *Off-Line Synchronization* on page 20. According to the systems business logic, versions identified as causing a global conflict must be invalidated to allow the resolution to occur without human intervention.

6.2.3 Validity in Software Development

Occasionally during software development source code versions may be introduced that either fail to meet some acceptability standards, such as not compiling or not passing all tests, or are otherwise not intended to describe a stable state of the project, such as an algorithm's partial implementation.

Some of reasons for invalidating a software's version include:

- Fault Introduction. A *fault* is a programmatic mistake introduced into the source code, that when executed may produce an error or unintended behavior [Sch05]

[IEE90]. A fault is generally perceived as a static characteristic of the source code but may not be immediately apparent to the software developer or tester. Any version that is identified as introducing a fault into the source may be considered as invalid.

- Unit Testing Failure. A *unit test* is an automated, low-level test of a particular behavior within code whose success or failure validates that unit of code [Ham05]. These tests are typically executed by the developer after the source code has been compiled to ensure that the software behaves correctly at the most primitive level. Any version that does not pass all unit tests may be considered invalid.
- Build Break. A *build* is the automated process of transforming source code into an executable product [Mec05] [LH07]. It typically is composed of a sequence of steps that include: (1) compiling, (2) testing, (3) packaging and possibly (4) deploying the resulting product. If the automated process is unable to successfully conclude any of the defined steps than the version may be considered invalid.
- Quality Assurance Team Testing Failure. A Quality Assurance (QA) team is responsible for performing a number of tests to ensure that the product will be acceptable with regards to the customer's expectations [Sch05]. While unit tests focus on the low-level functionality and are typically executed by the software developer. This testing is executed by a specialized team an focuses on the integrated product's functionality visible to the user. If a version is identified as not meeting the expectations of the user, it may be retrospectively classified as invalid.

  However, in an incremental developmental process, initial versions are not expected to provide the complete set of desired functionality. Therefore the measure of user's expectation should be adjusted to match the functionality designated to be present within the incremental version.
- Partial Implementations. Often when implementing particularly complex behaviors or making *big refactorings* [Fow99], a software developer follows a series of steps to reach the targeted final goal. The result of these steps are intermediate states which lie between the initial state and the desired state and do not represent an acceptable state of the source code.

  However, if the developer is forced to only move from one acceptable state to another and not allowed to record the intermediary steps, the danger of a major loss of work increase. The amount of work lost increases linear with the amount of effort expended. Therefore, it may be important to intentionally record versions of software that are not acceptable as a way of preventing the loss of work. This concept is similar to and provides the same basic support as checkpoints within a transaction [HR83].

It should be noted that this is not a exhaustive listing of all the reasons a software version may be considered invalid.

Intension and Observation

Interesting when considering these differing causes for invalidity, is that each is defined at a different point during the developer's work and may be intentionally or unintentionally invalid. A partial implementation commit represents an version that is intentionally invalid and the validity can be designated before the work is started. Validity definitions based on the compilation can be determined when the project is compiled and would result generally in an unintentional invalidity. Validity definitions based on testing also result in unintentional invalidity. However, depending on the level of testing that detects the failure, the definition of validity may or may not be directly perceived by the developer. If the developer detects the failure during the execution of unit tests prior to committing the state, the invalidity of the state is observed by the developer. However, if the version fails the tests of a QA team, it may retroactively be classified as invalid and this invalidation may not be observed by the developer. The differing times and intentions of determining validity is depicted in Figure 6.6.

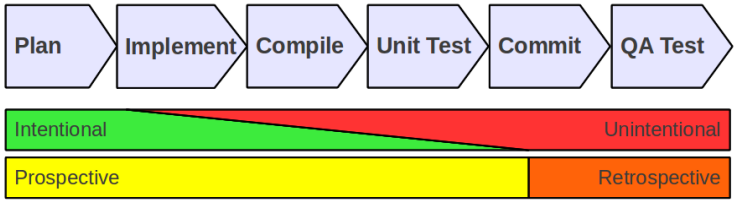


Figure 6.6: Determination of Version Validity

6.2.4 Definition of System and Valid Path

The system path represents a chronology of the sequence of changes applied and provides a complete representation of the history. This is the type of history managed and manipulated by contemporary VCSs, that lack the ability to differentiate between valid and invalid states. The valid path provides a limited chronology of the sequence of changes that represent actual forward progress through an history. The changes represented the valid path are a subset of the changes maintained by the system path, and are included in the valid path based on their validity.

Figure 6.7 on the next page provides a visual representation of the relationship between the system and valid paths. According to Equation 6.6 the system path (SP) is defined as the total number of committed states.Equation 6.7 on the next page defines the valid path (VP) as the complete subset of those states that are valid.

$$SP := \{v_{s0}, v_{s1}, ...v_{sn}\} \quad : \text{ n is total number of commits} \tag{6.6}$$

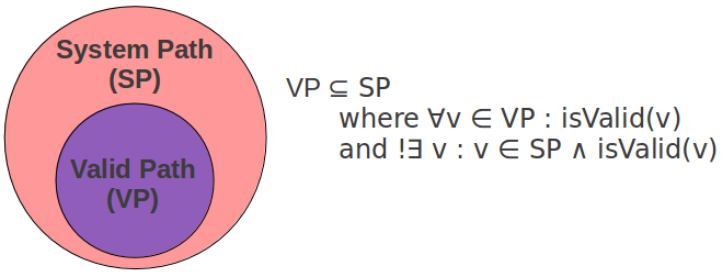


Figure 6.7: System and Valid Path Visual Definition

$$VP := \{v_{v0}, v_{v1}, ...v_{vm}\} \\ : isValid(v_{vx}) \wedge v_{vx} \in SP \text{ and } !\exists v : v \in SP \wedge isValid(v) \wedge v \tag{6.7}$$

The terms system and valid path will be defined as follows:

System Path – The complete set of recorded versions within a system that does not differentiate versions based on their validity characteristics which presents a complete non-monotonic depiction of a history.

Valid Path – A validity-based selection of the recorded versions which presents a predominately monotonic forward progressing depiction of a history by eliminating invalid versions from consideration.

The valid and system paths provide two differing perspectives of the same history. One includes all recorded versions while the other includes only those considered valid. Therefore, the valid path can be considered as a restrictive view of the system path and should always be maintained with the context of the overall system path. This allows both perspectives to be unified into a single context.

6.2.5 Benefits of Validity Tracking

The maintenance of each version's validity provides a number of benefits that can be categorized into three areas: (1) reduced search and analysis effort, (2) depiction of the predominately forward progressing history and (3) extension of the system path.

- Reduction of Search and Analysis Effort. When searching for an appropriate version to consider for an operation, much extraneous effort must first be expended to determine if the version is valid prior to effort expended investigating its specific characteristics. Working from a inherently invalid state only makes the effort more difficult. The valid path, removes this extraneous effort by removing invalid versions from consideration and allows the developer to focus on the key characteristics sought.

- Depiction of the Forward Progressing Evolution. The valid path provides a predominately monotonically forward progressing view of a history. This provides a better description of the actual useful work executed to produce a product and provides a good example of how the product may be correctly developed.
- Extension of the System Path. Since the valid path is defined in terms of the underlying system path, all functionality available in contemporary VCSs remains. The valid path concept provides an extension of the basic history analysis and manipulation functionality associated with VCSs.

6.3 Non-Conventional Means of Support

In this section techniques for supporting multi-headed versioning and validity tracking by employing common VCSs in non-conventional ways and their shortcomings will be considered. First, the multi-headed versioning with support for the autonomous versioning of each independent subcomponent (i.e. logical unit versioning) will be covered. Then, the ability to distinguish versions based on their validity properties and depiction of the valid path will be investigated.

6.3.1 Logical Unit Support

No VCS known to the author provides explicit support for maintaining an independent history for separate logical units or subcomponents that all belong to the same project, which maintains an independent history of the overall project's history. In general there is only a single history recorded per repository. However, a similar effect can be accomplished by employing and managing most common VCSs in non-standard techniques. Two common techniques are the use of subcomponent encapsulation, either in subrepositories or subdirectories, or completely separating each subcomponent's development into its own repository.

Separate Repositories

The simplest means of ensuring versioning autonomy for each subcomponent is by completely separating each subcomponent's development into its own repository and ignoring their interdependencies. In this technique there is no direct means for maintaining a state of the overall system. This technique provides good support for the independent development of components that exhibit no or very little coupling. However, there is no concept of the overall system state and no way of maintaining the interdependencies needed to reproduce a coherent system state.

Subdirectories

Most VCSs allow a single artifact or directory to be set to a specific historical version. In the subdirectory technique, the subcomponents are distributed into subdirectories. In

order to set a specific subcomponent to a specific version, the encompassing subdirectory is manipulated through partial reverts or checkouts. This allows the state of the overall system to be broken into subcomponents and differing versions of each subcomponent may be independently attained. This is the simplest technique that provides an overall system state, but has no direct support for the individual subcomponent states.

Subrepositories

Subrepositories refer to the technique of maintaining repositories within other repositories, creating a hierarchy of repositories. It is similar to the separated repositories approach but includes a parent project repository that encapsulates the overall project's state. This allows each subrepository to be individually manipulated, while the parent repository may provide control of the unified project history. This allows each subcomponent to maintain its own independent history. Of the three techniques, this provides the greatest level of control over the overall system structure and state. However, it also requires the greatest level of involved managerial effort. This structure may be created using Git using the following steps:

1. Create a subdirectory for each subcomponent.
   proj > mkdir *subcomponent-name*
2. Initialize a new repository in each of the subdirectories.
   proj > cd *subcomponent-name*
   proj/subproj > git init
   proj/subproj > cd ..
3. Initialize project repository and ignore subcomponent repositories.
   proj > git init
   proj > echo ".git" >> .git/info/excludes

Once this structure is created, the developer may commit and revert the state of each subproject and the overall project independently by executing the appropriate command in the appropriate project or subproject directory.

Figure 6.8 on the facing page provides a visual comparison of the two encapsulating techniques. The section on the left depicts the technique of encapsulating repositories and the section on the right depicts the technique of encapsulating directories.

Problems with Non-Conventional Solutions

However, employing a system in non-standard ways for which it is not intended often brings with it inherent difficulties. While these techniques may function to some extent, they exhibit several critical weaknesses that are common to all techniques:

- Additional Management and Control Support. The user must either explicitly manipulate each separate repository/directory or an additional supporting system must be integrated to facilitate the independent manipulation. A simple set

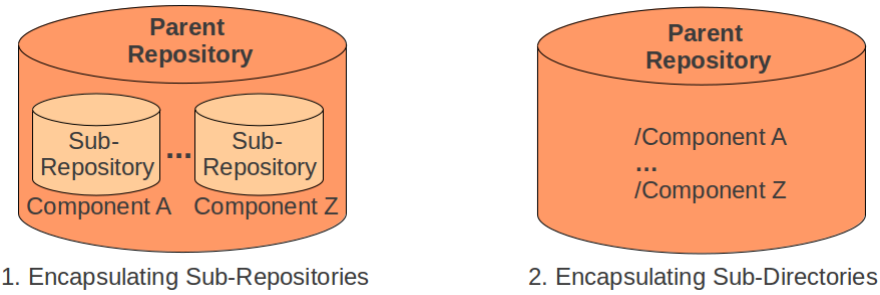


Figure 6.8: Encapsulating Subcomponent Techniques Comparison

of scripts may help alleviate some of these difficulties, but lack the coherency of an integrated system. The tracking and management of each separate subcomponent's versioning history drastically exacerbates the problem. The user must know or derive the exact version of each separate subcomponent to regenerate a desired state. This problem is somewhat alleviated in the subrepositories technique. However, this limits the set of reverted states the set recorded in the parent repository.

- No Shared System Awareness. Information may not be shared between parent-child or sibling repositories. A parent repository perceives the subrepository simply as another set of files. Likewise, the subcomponents means of interacting and exchanging information with the parent or sibling repositories. This limits the exchange of data and sharing of the workload and between repositories. Most importantly, it restricts the system's development and employment.
- Waste of Resources. Each repository requires a given overhead. Typically a single repository's overhead is minimal with respect to the overall project. However, each change to each subrepository is reflected in the parent repository. This doubles the effort of recording each change and in a system supporting a large number of subcomponent's this could have an impact.
- Artifact Version Transfer and Sharing. Since, there is no common awareness of the overall system's structure, artifacts and data may not be simply transferred from one subcomponent to another. Additionally, during the progression of a growing system it is inevitable that at some point a subcomponent may need to be split into multiple separately evolving subcomponent. It is difficult in any of these techniques to split the system while maintaining the previously recorded history.
- No IDE Integration. Little work is done in the field of software development without the use of a good IDE. IDEs are capable of leveraging VCSs in known standard ways. There is little support for these described non-standard employment techniques described.
- Pre-designated Directory Structure. These approaches do not allow files from different subcomponents to be maintained in the same directory. Each component

must be strictly separated into its own directory.

6.3.2 Valid Version and Path Support

Contemporary VCSs provide no explicit means for designating, maintaining or traversing a history's valid path. However, three techniques may be applied to provide a similar effect: (1) properties, (2) cherry picking or pre-tested commits and (3) branching.

Explicit Validity Properties

Interestingly enough, support for this concept was included as a key component in both the SCCS and RCS. SCCS's special deltas and RCS's version attributes where provided to support this functionality. In both of these approaches properties are explicitly associated with a version and the validity of a version may be determined by querying these properties. While this concept is no longer plays a central role in most contemporary VCSs, most still provide the capability to associate user defined properties with specific states [PCSF08]. Figure 6.9 depicts the resulting view of an explicit property-based history. As depicted, the only way to traverse the history is along the system path.

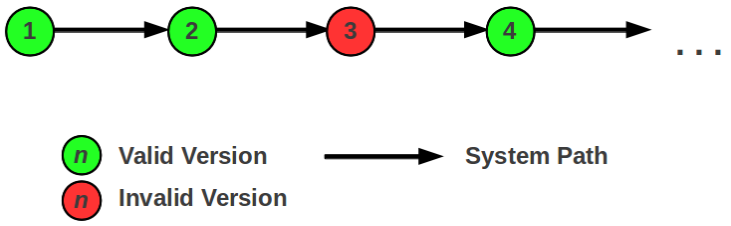


Figure 6.9: Explicit Properties Based Valid Path

However, in this approach there is no support for direct traversal of the valid path. The system path must be traversed and the validity of each commit must be queried. Additionally, the validity must be explicitly set by the developer.

Branching

This approach to validity tracking is similar to the properties-based validity definition, but attempts to provide primary support for traversal along the valid path instead of the system path. Here invalid versions are introduced as a branch associated to the previous valid version. Figure 6.10 on the facing page depicts the resultant view of the history-based on the branching method. Here the primary means of traversal is along the valid path.

There are three key challenges to this technique. First, each invalid version represents a validity branch. Supposing over a project's development thousands of commits are made and ten percent are for some reason deemed invalid. This would result the creation of hundreds of branches that must be managed. Secondly, traversal along the system

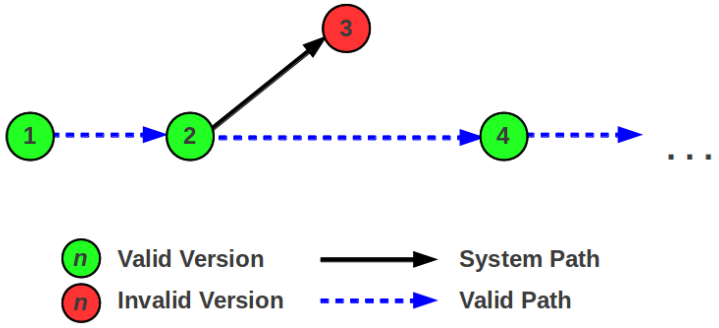


Figure 6.10: Branching Based Valid Path

path involves finding the next version, regardless of validity. A generalized means of determining if the next version is valid or invalid remains difficult. Finally, if a version is later determined to be invalid that portion of the history must be explicitly manipulated to reflect its new structure.

Cherry Picking and Pre-Tested Commits

Recently, validity tracking has drawn renewed attention [Loe09] [MTM+07]. Cherry picking and pre-tested commits are two similar efforts that restrict the introduction of commits to a repository based on their perceived acceptability. This process ensures the developmental work is progressing along a solid foundation of work and all versions represent a deliverable product.

Cherry picking [Loe09] is the process of selecting specific acceptable version for introduction into an history. Generally, it takes a specified commit in one branch and introduces it into another branches history, creating a new commit. An example of this technique in action is the actions of the integration manager or lieutenants in the distributed collaboration workflows described in Section 5.5.1: *Collaboration Workflows* on page 36. Here the responsible party selects the acceptable versions from a public repository and introduces them into the blessed repository. This effort results in a valid path being maintained in the blessed repository while the combined histories of the various public repositories could be considered the system path.

Pre-tested commits, as discussed in [Jet11] [Hud11] [Wor11], is an attempt to automate the cherry picking process based on a test result selection criteria. When a developer introduces a new commit, it is not automatically integrated into the blessed repository. Rather it is committed to a intermediary repository and all automatic tests are ran against the new version. If all tests success, the version is applied to the blessed repository and is made available to other developers. Otherwise a notice is sent to the developer. In this manner, the intermediary repository maintains the system path and the blessed repository maintains the valid path. Figure 6.11 on the next page depicts the workflow involved in the cherry picking or pre-tested commit development.

However, this approach has a number of weaknesses. First, it hides all non-valid

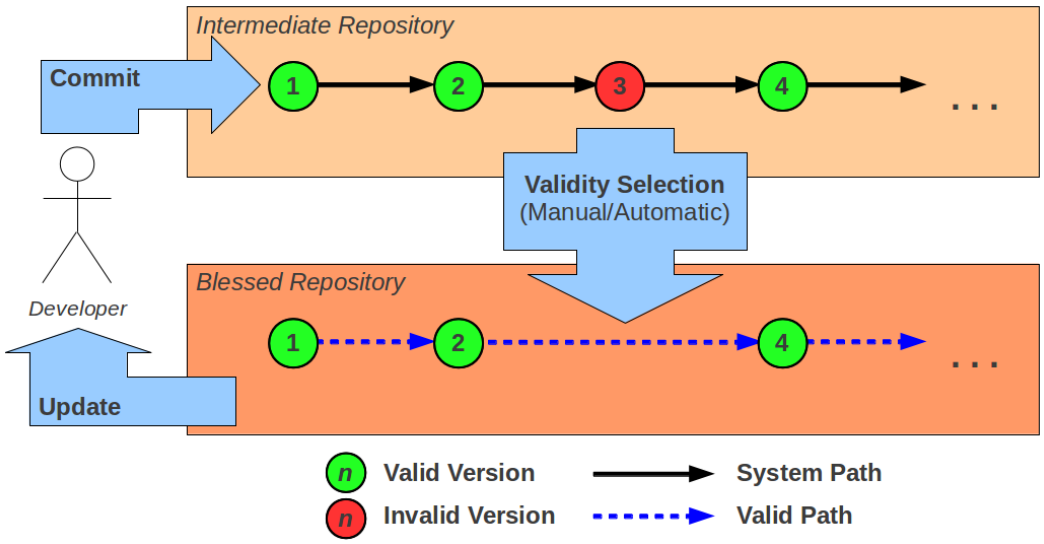


Figure 6.11: Blessed Repository Valid Path

commits from the developers. This prevents the ability to learn from past mistakes and the possibility of gleaning correct portions from an overall invalid version. Additionally, it introduces the overhead associated with the management of two or more separate repositories. All valid versions are persisted twice; in the intermediate repository and in the blessed repository. Furthermore, the committed work of one developer is not immediately made available for usage by other developers. This may introduce an unacceptable delay in the developmental cycle, depending upon the delay associated with the validity selection.

Problem with Non-Conventional Solutions

While these techniques may support the basic concepts of the valid path, but each is hindered by a number of challenges. Challenges specific to their approach were described in the previous sections. However, there are a couple of other challenges that are common to all of these techniques:

- Static Definition of Validity. All of these techniques perceive the valid path and system path as static descriptions of a history. Any changes to the history require a review of all of the persisted versions to determine the impact of the change. The validity of each version must be explicitly determined and set. There is no means of deriving the validity of a state from the historical context.
- Preference of System or Valid Path. Each technique exhibits a prejudice towards either the system or valid path. However, both paths are important and provide unique and equally important perspective of the history. The prejudices result in systems that are not flexible and not appropriate for employment in one or the other type of environment.

6.4 Alpha-Flow Adequacy

In this section, the adequacy of the various VCSs will be considered with respect to the versioning needs and goals of the α-Flow project. The VCSs, Subversion, Git and Mercurial, will be considered. Table 6.2 depicts the key characteristics of the VCSs and classifies them as appropriate (green), acceptable (yellow) and not acceptable (red).

	Subversion	Git	Mercurial
Non-Proprietary	Yes	Yes	Yes
Platform Independent	No (C/C++)	No (C/C++)	Partial (Python/C)
Executable Size	7.5 MB	19.1 MB	4.5 MB*
Repository Size	Large	Small	Medium
Distributed	No	Yes	Yes
Rebasing	No	Yes	Yes**
Multi-Headed	No	No	No
Validity Tracking	No	No	No

* without rebasing extension

** with rebasing extension

Table 6.2: Alpha-Flow Adequacy

As depicted, none of the VCSs are perfectly suited to meeting α-Flow's versioning needs. Mercurial appears to be the most appropriate of the compared VCSs, but it also lacks in the areas of platform independence, rebasing capabilities and executable size. Furthermore, none of the compared VCSs provide explicit support for multi-headed versioning or validity tracking.

This inability to adequately support the versioning within α-Flow introduces the need for a new VCS. This VCS will be a multi-headed VCS that supports the tracking of versioning validity. It will be non-proprietary, platform independent, have a very small executable and repository size with respect to the information under version control. Finally, it will provide a means for reorganizing an artifact's history.

6.5 Summary

In this chapter the concepts of logical units and version validity were initially introduced and refined through their consideration within the context of α-Flow and software development. Logical units are independent sets of artifacts that exhibit loose coupling to other artifacts not included within the set and high intra-set cohesion. They also may require or benefit from an independent versioning. Validity is a property expressing the application-specific correctness or acceptability of a given version.

Next, techniques for supporting these concepts within contemporary VCSs and their shortcomings were considered. Finally, the VCSs under consideration for supporting α-Flow's versioning needs were analyzed and were rejected as being inadequate. This

provides the grounding of the need for a new VCS, hydra, which will support these concepts and adequately fulfill the versioning needs of the α-Flow system.

7 Design – Versioning Core

In this chapter a Multi-Headed Version Control System (mVCS), Hydra, capable of validity tracking and supporting α-Flow's versioning needs is designed. First, the internal core and versioning model will be considered and designed. Next, the extension of the versioning model to support multi-headed versioning and validity tracking will be introduced. Finally, the concept of content addressable storage is extended to improve support for history alterations. Through these steps, the internal core for the Hydra will be designed.

This chapter focuses exclusively on the versioning core. The persistency and differential subcomponents, while important, do not directly contribute to the versioning functionality of this version of the system and will be discussed in Section 8.2: *Persistency Subsystem* on page 100 and Section 8.4: *Differential Calculation* on page 106.

7.1 Versioning Core

In this section the core of Hydra will be designed. First, the versioning model will be designed and then its assumed repository structure will be considered. The versioning model defines the key components that will collaborate to accomplish the system versioning requirements. The repository, which is responsible for persisting the information required to reproduce a persisted state, must then be organized in a manner that best suits the envisioned versioning model.

7.1.1 Versioning Model

To develop the versioning model the following steps will be taken. First, the key roles and their purpose will be introduced. Next, the relationships between and responsibilities of these roles will be defined. Finally, the commonalities shared by these roles will be extracted and an initial class diagram will be produced.

Role Definitions

The versioning model is the central concept upon which the VCS will be built. It is responsible for representing: (1) the files under version control, (2)their organization and (3) their recorded versions. These three elements represent the key actors within the versioning model and are given the names *artifact*, *container* and *state* respectively.

artifact represents an element under version control. Artifacts are organized into interrelated groups and changes are recorded as new versions.

container provides the means to organize the artifacts into groups, similar to how folders within a file system are used to organize files.

state represents a snapshot of a set of artifacts contained within a container. While a version emphasizes a single artifact, the state emphasizes the collective set of artifacts. It provides the means to record and maintain the interdependencies between the collective set of artifacts.

This versioning model draws heavily from Git's versioning model. Git's model provides a clean and simple organization that represents these various abstractions. However, the names used in Git's versioning model, i.e. blob, tree and commit, reflect their technical implementation details and not their purpose. Defining actors according to their roles or purpose provides a higher level of abstraction and allows the underlying implementation to be changed without confusing the model. Thus, the names used in this model are chosen to emphasize their purpose and not implementation.

The name *blob* indicates that it represents some Binary Large Object (BLOB). However, the purpose of this element is to represent a file under version control. This includes not only the file maintained within the workspace, but also a reference to the recorded versions of that file. Therefore, the term *artifact* is used to describe an file under version control.

The name *tree* represents the resultant structure created by this element's recursive structure. However, the purpose of this abstraction is to organize a collection of artifacts into a group or set. Therefore, the term *container* is used to describe an element that contains other elements and provides a generalized organizational structure.

Finally, the name *commit* represents the act of persisting a given version and not the information persisted. In this versioning model, a snapshot of the various artifacts' versions is persisted to maintain the inter-artifact dependencies. The information contained within the snapshot, i.e. the artifact interdependencies, and not the act is important. Therefore, the term *state* is used to describe a specific snapshot of the artifact versions. Figure 7.1 is a depiction of the Class-Responsibility-Collaboration (CRC) Cards for the artifact, container and state roles.

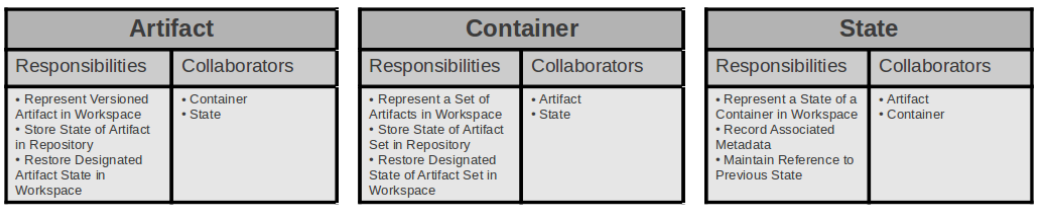


Figure 7.1: Artifact, Container, and State CRC Cards

Role Relationships

In this section the relationships between the previously defined roles is considered. First, the relationship between an artifact and a container will be analyzed. A container provides a means of organizing or grouping a set of artifacts. Logically deducted from this definition that a container maintains references to any number of artifacts. However, similar to a folder hierarchy, a container may also contain any number of subcontainers. This creates a flexible recursive structure that can be used to organize a set of objects in arbitrary ways.

Next the relationship between the container and state are considered. Of critical importance is the determination if a state may reference more than one container. Theoretically, it could be assumed that a state may reflect a snapshot of multiple sets of artifacts. This encourages a one-to-many relationship between the state and container. However, the container, like Git's tree, is a recursive structure. Thus, the architecture can be simplified by assuming a single root container provides the encapsulation of a more complex substructure and simplifies the state-container relationship to one-to-one.

Finally, the state must maintain references to its previous states in order to represent a history of changes made to a collection of artifacts. This creates a recursive structure also in the state. However, the number of previous states maintained must be considered. Maintaining only a single previous state creates a linear depiction of the evolution but is unable to accurately depict the merging or other situations. Therefore, increased flexibility is allowed by supporting the referencing to an arbitrary number of previous states. The resulting versioning model is depicted in Figure 7.2.

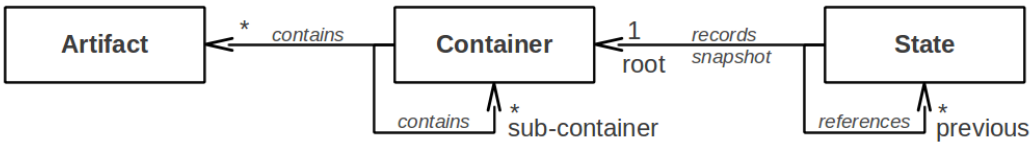


Figure 7.2: Artifact, Container and State Role Relationships

Class Definitions

In the last sections the roles have been identified and their purpose and relationships have been investigated. Now a concrete definition of these roles as classes will be introduced to provide the details necessary for their implementation.

The artifact represents an element under version control. Its behavior, drawn from the role's responsibilities, consists of two functions: (1) storing of the contents of the represented artifact, its version, at any given time and (2) returning the contents of the represented artifact to a previously persisted state. These operations are given the names store and retrieve. As the content is being stored in or retrieved from the repository.

The container represents a collection of artifacts and subcontainers. It must manage, store and retrieve the represented set of artifacts as a coherent set. Managing a set of artifacts requires the ability to add, remove and query the set. Storing and retrieving the artifacts can take advantage of their inherent ability to store and retrieve a designated state. The container is responsible for coordinating the actions of the managed artifacts.

The state represents a snapshot of a collection of artifact versions. Its primary behavior is the storage and retrieval of a specific state from the repository. Additionally, it is responsible for maintaining the associated metadata and references to the states from which it originates. The metadata consists of the party responsible for creating the state, the state's purpose and when the state was created. In order to maximize flexibility, the state will provide the ability to alter its set of previous states. It will be able to add, remove and query its set of previous states.

Through an analysis of the behavior of these three key elements, it may be seen that they exhibit similarities. Each is capable of storing its current content and retrieving a persisted state. These similarities may be extracted to a common superclass, the Element. This extracted superclass encapsulates the conceptually common functionality of storing and retrieving their respective data. The classes Artifact, Container and State are all specifications of an Element within the versioning model. Figure 7.3 is a class diagram that depicts the versioning model described in this section.

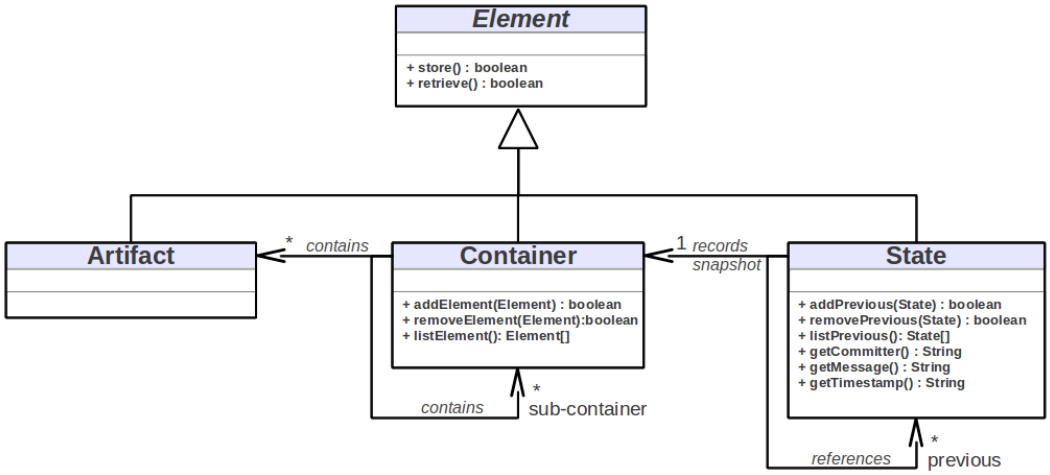


Figure 7.3: Versioning Model Class Diagram

7.1.2 Repository Design

VCSs are broken into two component areas: (1) workspace and (2) repository. The workspace maintains a private working copy of the artifacts which are visible to and editable by the user. The repository is a database that maintains all persisted versions and any organizational information necessary to support the VCS's operations. As described in the previous sections, a VCS is primarily responsible for persisting artifact versions

into the repository and restoring a user specified artifact version to the workspace. The interaction of these basic component parts is depicted in Figure 7.4.

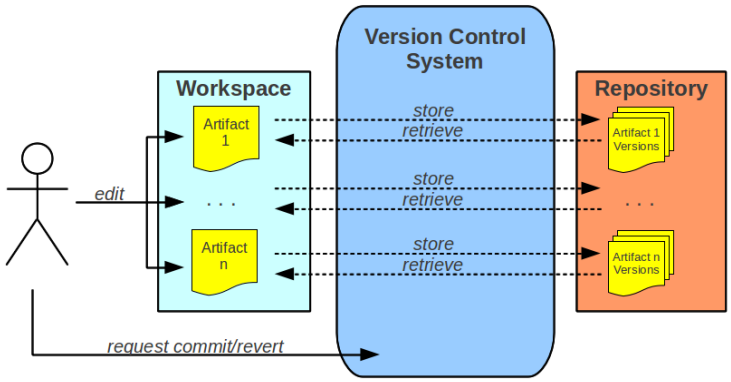


Figure 7.4: VCS Component Parts

A VCS's primary responsibility is the transfer of the requested data between the workspace and repository. In order to accomplish this task efficiently, two critical concerns must be optimized: (1) data storage and (2) data transfer. First, the data must be stored in a manner that optimizes the ability to find and regenerate any designated version. Second, once found the designated version must be transferred to the designated location.

The task of transferring data from one location to another plays a critical role when considering the performance of the VCS, but has no significant impact on the core versioning model and thus will not be further discussed in this chapter. Section 8.2: *Persistency Subsystem* on page 100 provides an in depth analysis of this functionality. The data storage plan does have a direct impact on the versioning model and will be further discussed.

There are two areas of concern when considering data storage. First, in what format is the data to be stored? Second, how is the persisted data organized in the repository? The first affects the resultant size of the repository and the time it takes to recreate a designated version. The second affects the speed at which a designated version can be located.

Version Storage Format (Differential vs. Intact Copies)

Versions are typically stored either as differentials or as complete copies. Differential storage is generally assumed to reduce the size of the storage space required but increase the time required to retrieve an arbitrary version. Complete copy storage is generally assumed to require more space but less time is required to retrieve an arbitrary version.

Size is the primary concern for α-Flow and thus seems to encourage a differential approach. However, the benefits of differential storage are dramatically influenced by three aspects: (1) expected document formats, (2) relative size of documents and (3)

number of expected versions. Consideration of these aspects with respect to α-Flow indicates that differential storage may not be appropriate. These three aspects are as follows:

- Expected Document Formats. α-Flow expects to deal with a wide variety of document formats. The majority of these documents formats are either non-text based, e.g. PDF, or of a proprietary format, e.g. Microsoft Word Document. These formats are generally not conducive to the calculation of differentials. Therefore, a relatively small portion of the documents can even be considered to benefit from differential storage.
- Relative Document Size. The typical text-based document that is conducive to differential calculation is of minimal size, i.e. 10-100 KBs. Meanwhile, binary and other proprietary formatted documents are of a size 10-100 times greater than the text-based documents. Thus, any benefit gained by differential calculation would be minimalistic when considering the overall storage requirements.
- Number of Expected Versions. For each document, at least one full copy must be maintained. Thus, benefits from differential storage are dependent on the number of versions maintained. If only a few versions are expected for each artifact, the benefits are meager. Within α-Flow each α-Card represents a single activity which is generally only executed once and only a small quantity of versions are expected to be created.

In general the documents, for which a differential may be calculated, represent a relatively small percentage of the space required within the α-Flow system. The majority of the space is taken by other documents not conducive to differential storage. Additionally, few versions beyond the initial version are expected. Therefore, it can be assumed that differential storage is not the appropriate form for storage. Rather, complete copies should be stored and the developmental effort should be spend attempting to find ways to compress the storage in other ways. Means of compressing the persisted data is further discussed in Section 8.2.5: *Storage Strategies* on page 105.

Version Storage Organization (Hierarchical vs. Content Addressable Storage)

The organization of the storage heavily influences the characteristics of the VCS. It affects the rate at which arbitrary versions can be located as well as the size required to store all versions. Two techniques for data storage and access are: (1) hierarchical and (2) content addressable storage. Hierarchical storage is similar to that perceived on a common file system where files are located based on their position within a folder hierarchy. In content addressable storage there is no hierarchical organizational structure, rather information is directly addressed based on a hashing of their contents.

Forms of hierarchical storage can be organized according to their focus: version or artifact. Within version focused storage, the hierarchal structure is first organized according to the recorded versions and then the artifacts. SVN is one example of version

focused storage [PCSF08]. Within artifact focused storage, the hierarchal structure first breaks on the artifacts and then further according to the artifact's versions. CVS is one example of artifact focused storage [Gru86]. These two organizational structures are depicted in Figure 7.5.

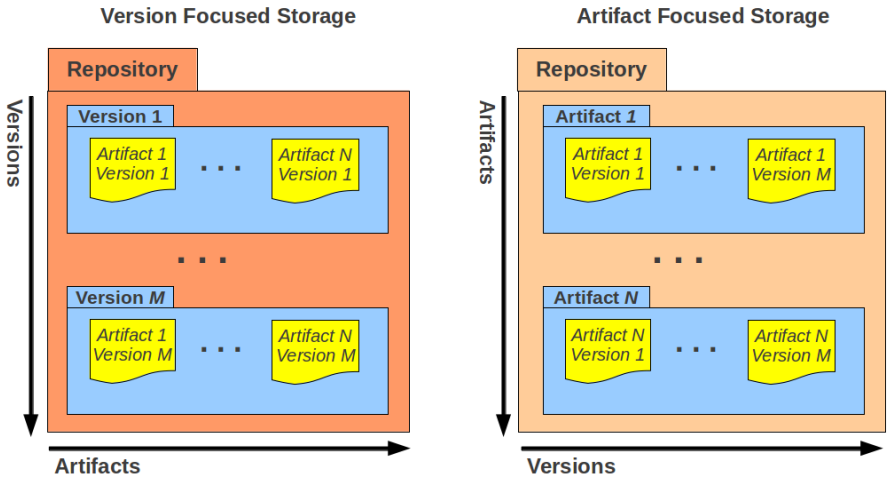


Figure 7.5: Hierarchical Storage Organization Comparison

When considering the depicted storage structures, it can be observed that the versions are organized into a grid pattern in both techniques. Thus, the location of any specific version of an artifact requires a search along two axes: artifact and version. The version axis is preferred in the version focused storage and emphasizes the overall coherent version. Oppositely, the artifact axis is traversed first in the artifact focused storage and this emphases the evolution of the individual artifacts. These storage structures present two difficulties. First, a search along two axes is required to determine a specific version of a specific artifact. Within a folder hierarchy this could result in an expensive search operation. Secondly, each version of each artifact is explicitly stored. Even if two versions are equivalent, they are both required.

Content addressable storage provides a means to alleviate these two problems. Content addressable storage stores data not in a folder hierarchy but in storage slots or locations according to its content [Loe09]. Thus the search for a specific version is reduced to a direct access based on its content and two versions that have the same content will always be stored in the same space, regardless if they come from the same artifact or a different artifact.

Fingerprints – Uniquely Identifying Content

A Unique Identifier (UID) calculated from a given content is necessary to define its storage location. Conceptually, this unique identifier is similar primary key which provides a logical reference within a database. The UID's calculation plays a critical role in this storage technique. It must guarantee that unique content creates a UID and like

content will always generate the same UID. Additionally, it must be efficient, as all data accesses will require its calculation.

In order to support the content addressable storage, each Element must be able to generate a UID based on its content. This introduces a new role to our versioning model: ***fingerprint***. Fingerprint is responsible for calculating and maintaining the UID based on given content and one is associated to each distinct Element within the system. The definition of the various elements content requires some consideration.

The content associated with an Artifact must reflect the contents of the referenced workspace file. The content associated with a Container must reflect the collective set of contained artifacts. The content associated with a State must reflect the combination of the snapshot of artifact versions and the associated metadata. This requires two separate means of calculating the unique identification: a file based method for an Artifact and a dynamic string based method for Containers and States. The updated class diagram, depicted in Figure 7.6, reflects the introduction of the Fingerprint class which is used to uniquely identify each Element based on its contents.

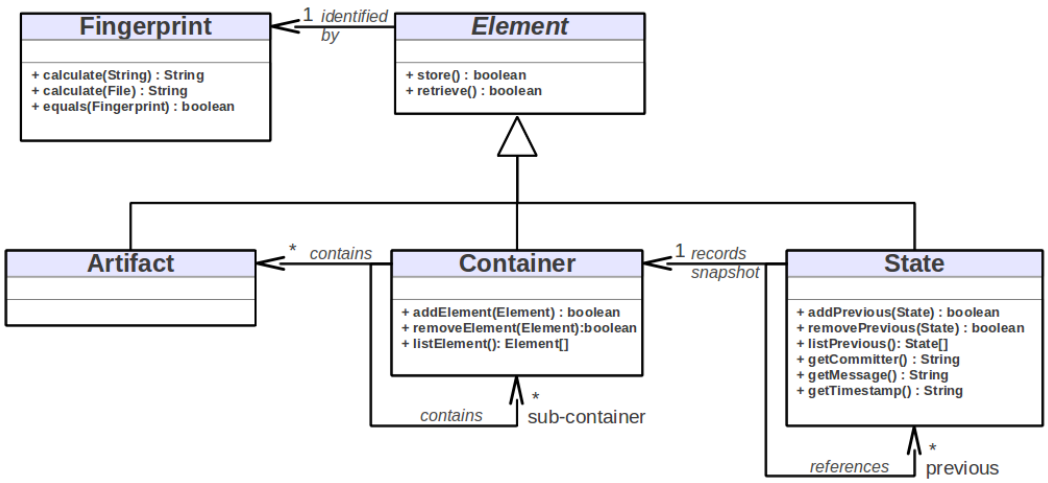


Figure 7.6: Updated Versioning Model Class Diagram Including Fingerprint

Artifact Content Format

As described in the previous section, each element of the versioning model will be persisted and retrieved based on a unique identifier calculated based on the content. The fingerprint is responsible for providing this capability. Its specific implementation details will be further refined later in Section 10.2.1: *Fingerprint Calculation* on page 114, but it can be assumed that the fingerprint will employ some hashing function, such as SHA-1, to accomplish its goals.

The content of an artifact is exclusively the content of the referenced workspace file. It does not include the files name nor any of its metadata, i.e. last modified date. This maximizes the reuse of persisted content and allows two files with different names to

share a single stored instance of their content, reducing the space needed. Figure 7.7 depicts the format of information used to calculate an artifact's fingerprint.

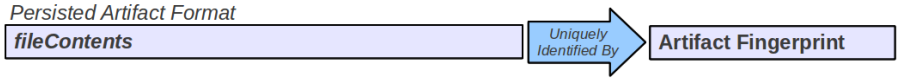


Figure 7.7: Persisted Artifact Format for Calculating Fingerprint

Container Content Format

The container is responsible for organizing a set of artifacts and subcontainers. Each artifact maintains its own unique fingerprint, but not its metadata. Therefore, the container is responsible for associating an artifact's content to a given name and location. Subcontainers must also maintain their own unique fingerprint; which can be assumed to be recursively calculated.

Figure 7.8 depicts the format of the previously described information that is used to calculate a container's fingerprint. Additionally, this dynamically derived string contains all information necessary to represent a container's state and will be stored within the repository in a location identified by its fingerprint. Characters that are not italicized are to be written explicitly, while a set of italicized characters indicates information to be dynamically determined. An asterisk indicates that zero to n instances of this information may be present and curved braces indicate information that may or may not be present. Each line represents a specific piece of information. The set of characters ::>> is used to delineate portions of the information on a single line. The purpose of each lines information may be identified by the initial two character abbreviation. HH stands for "Hydra Header", CO stands for "Container" and AR stands for "Artifact". The first line is necessary to provide a unique identification for an empty container. Each subcontainer is depicted as an individual line which contains both its name and its content hash, which may be used to access its content. Each contained artifact likewise occupies its own line which maintains both its name and its content hash.

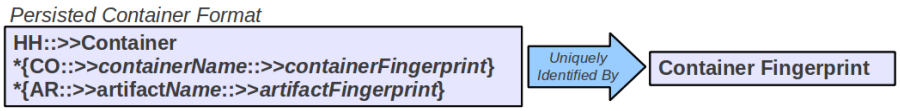


Figure 7.8: Persisted Container Format for Calculating Fingerprint

State Content Format

The state is responsible for maintaining a specific snapshot of the root container, the associated metadata and references to all previous states from which it was derived. The snapshot of the root container may be assumed to be of the format employed to define

a subcontainer. The associated metadata consists of the committer's identification, the timestamp indicating the time of its creation and a message indicating the purpose of the state. Unlike the artifact and container, a state has no associated name. Therefore, the listing of previous states may be recorded as a list of their fingerprints.

Figure 7.9 is a depiction of the format of the previously described information that will be used to calculate a state's fingerprint. Similar to the container's format it has a header identified with the *HH* abbreviation. The root container's name and fingerprint is maintained in the second line identified by the abbreviation *CO*. Next any number of lines, identified by their *PS* abbreviation, is used to maintain a state's references to its "Previous States". The state's "Metadata", identified by the *MD* abbreviation, maintains the responsible committer, a timestamp indicating its creational time and a message describing the state's purpose.

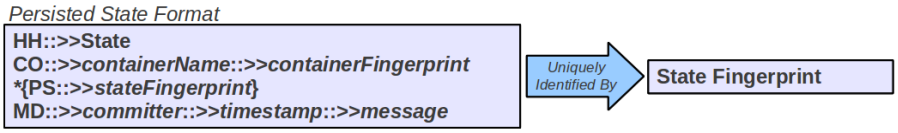


Figure 7.9: Persisted State Format for Calculating Fingerprint

Repository References Example

To better understand how the relationships between the various versioning elements are maintained within these formats, an example is presented. Assume a folder hierarchy with three directories: *Directory1*, *Directory2* and *Directory3*. *Directory1* contains *Directory2*, which in turn contains *Directory3*. Additionally, assume three files: *File1*, *File2* and *File3*. *File1* and *File2* are contained within *Directory1* and *File3* is contained within *Directory2*. Further assume that each of these Elements has a distinct Fingerprint: *fpD1-3* for the three directories and *fpF1-3* for the three files. This structure and respective fingerprints are depicted in Figure 7.10.

Directory Structure	Fingerprints
- Directory1/	→ fpD1
- File1	→ fpF1
- File2	→ fpF2
- Directory2/	→ fpD2
- File3	→ fpF3
- Directory3/	→ fpD3

Figure 7.10: Example Directory Structure and Fingerprints

To record a coherent state of the artifacts, the fingerprint relationship between the various elements must be maintained. This is accomplished through the previously

described fingerprint formatting. Figure 7.11 on the next page depicts a concrete example of how a coherent state for the previously described structure is maintained. The State references the root directory, *Directory1*, which maintains the references to *File1*, *File2* and *Directory2*. *Directory2* then maintains the references to *File3* and *Directory3*. *Directory3* is an empty directory and therefore maintains no further references.

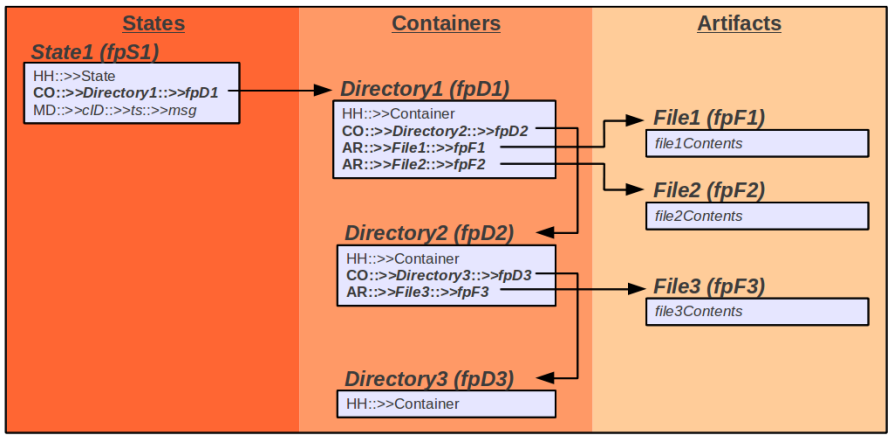


Figure 7.11: Versioning Model Format References Example

One last important consideration is the size of these formats. Each format reflects a size of approximately 200 Bytes. When considered with respect to the size of the files under version control, which are typically much greater than 1 KB, this extra space is minimal.

7.1.3 Versioning Example

In this section two examples will be presented to visually depict the how the versioning model maintains fingerprint references while a new state is produced. Both show how persisted content may be shared by multiple elements based on their fingerprint, thus reducing the overall space required. These examples start with two files, *File1* and *File2*, located within the workspace, *Project*.

Figure 7.12 on the following page provides a visual depiction of the situation after an initial commit. The left represents the current workspace, in the middle is a depiction of the versioning model and on the right is a listing of the fingerprinted elements that are persisted within the repository. States are depicted as circles, containers as triangles, artifacts and files as files and folders as rectangles. Additionally, each of the elements will be annotated with their fingerprint. Bold annotations indicate a new fingerprint which is represented as a new entry within the repository. Outlined and shadowed annotations indicate elements that have the same fingerprint as another previously stored element and thus require no new storage space.

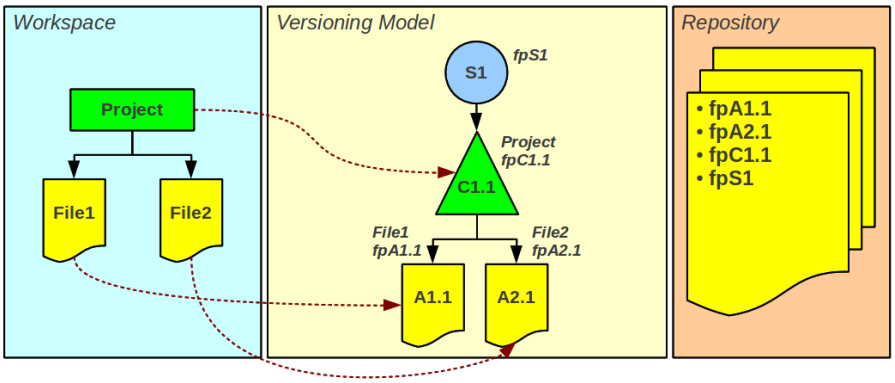


Figure 7.12: Versioning Examples Outset Situation

Unchanged Artifact References

In the first example, the sharing of persisted content between states in which the artifact does not change is examined. Assume that the content of *File1* is altered and a new state is committed. Figure 7.13 depicts the resulting situation. The artifact representing *File2* has not changed, and maintains a reference to the same persisted content, *A2.1*, in both states.

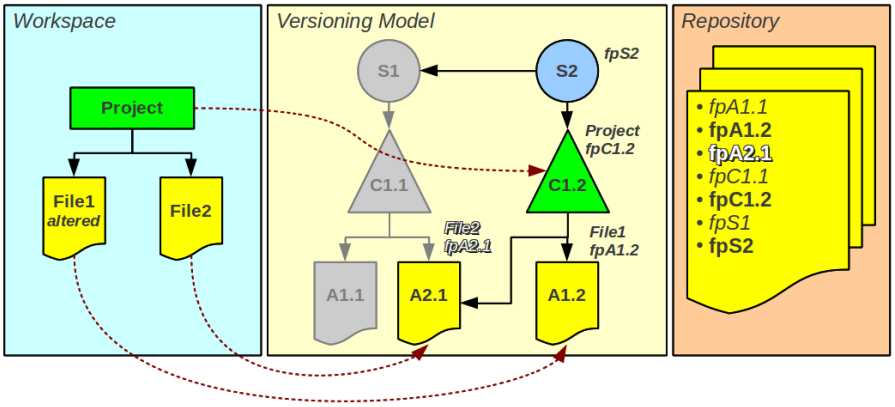


Figure 7.13: Sharing of References for Unchanged Artifacts

Moved/Copied Artifact References

In the second example, the sharing of persisted content succeeds from the moving artifacts from one place to another. Assume that a new subdirectory, *Sub*, is created within the directory *Project*. *File1* is moved to the subdirectory and a copy of *File2* is also moved into the subdirectory. After these changes are completed, a new state is recorded. Figure 7.14 on the facing page depicts the resulting state. Almost the entire

new state consists of references to previously persisted content. Only the new state and a new container were introduced into the repository.

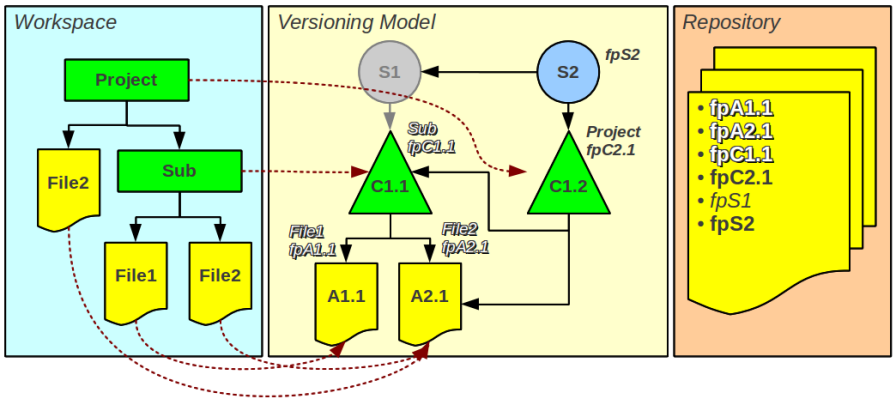


Figure 7.14: Sharing of References for Moved Artifacts

In these examples, two ways in which the content addressable storage technique reduces the overall space required are presented. This approach requires no additional space when an artifact or container, that has the same fingerprint as a previously persisted element, is stored.

7.2 Multi-Headed Versioning

In the previous section, the versioning core design was introduced. This section will describe the extension of the versioning model to support multi-headed versioning. Its success hinges on the use of the logical unit.

Logical units organize a larger project into independent units, whose history may be independently tracked. The logical unit's states maintain the intra-unit dependencies, while the project state maintains the inter-unit dependencies. Contemporary VCSs support only a single history which maintains both intra- and inter-unit dependencies. However, this forces each unit to be developed at the same lockstep pace, essentially eliminating the possibility for independent management and development.

First, the key roles, their purposes and their relationships will be introduced. Next, these roles will be refined and integrated into the versioning model. After extending the versioning core, the repository's design design will be extended, to support the new referential elements. Finally, the concept of committing will be reviewed with respect to the new level of versioning granularity.

7.2.1 Extension of the Versioning Core

In this section the extension of the versioning core from a structural and behavioral perspective will be considered. The intent is to extend the core class diagram presented

in the previous section to include the concepts necessary to support multi-headed versioning. The new actors and their relationships will be considered an then they will be integrated into the base class diagram. Finally the integrated class diagram will be analyzed to identify any possible commonalities that may be extracted into superclasses.

Role Definitions

As described in Section 6.1.4: *Logical Unit Definition* on page 48 a logical unit is a logically independent set of artifacts, that exhibits loose coupling to other artifacts and high intra-set artifact cohesion, which require or benefit from an independent management. This identifies and defines the purpose of the first role, logical unit, which is needed to extend the versioning model.

However, a logical unit only deals with the intra-unit dependencies and represents an independent subcomponent of the whole. Therefore, another role is needed to be defined to represent the overall project. Its primary purpose is to describe the integrated evolution of the defined logical units and accounting for the inter-unit dependencies. If the logical units are considered to be independent actors, their integration may be visualized as a theatrical stage. The individual actors have their own set of lines and could act independently of the others. However, their independent actions do not portray the overall story. The stage is where the individual actor's interact to present a performance. Therefore, the stage is the role that is responsible for managing the integration of the various logical units into a cohesive state. Figure 7.15 is a depiction of the CRC Cards for the logical unit and stage roles.

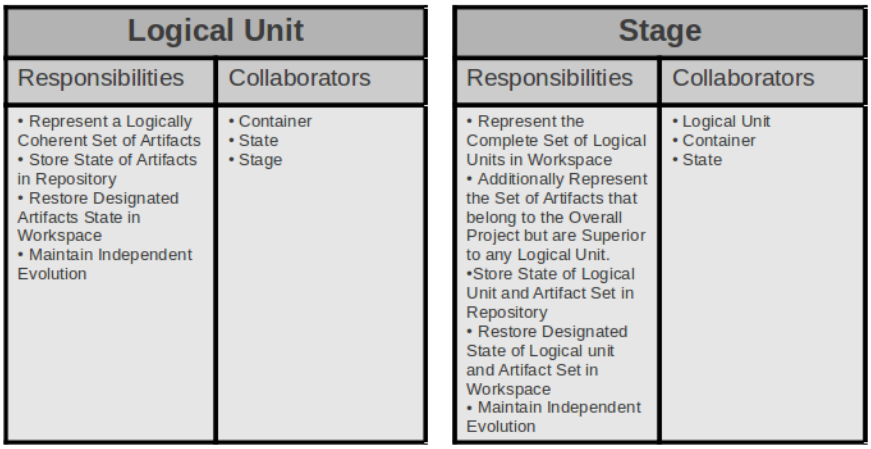


Figure 7.15: Logical Unit and Stage CRC Cards

Based on these two roles, it can be observed that each role is required to maintain a state. But each respective state has a different purpose. The logical unit's state is responsible for maintaining the state of a set of artifacts; while the stage's state is responsible for maintaining the state of a set of logical units. Thus, the singular state role must be divided into two separate roles: the *logical unit state* and the *stage state*.

When considering the responsibilities, the stage state is an extension of the logical unit's state. It provides the same capability of recording the state of a set of artifacts, but is also required to record the state of a set of logical units. Figure 7.16 depicts the respective state roles for the logical unit and stage.

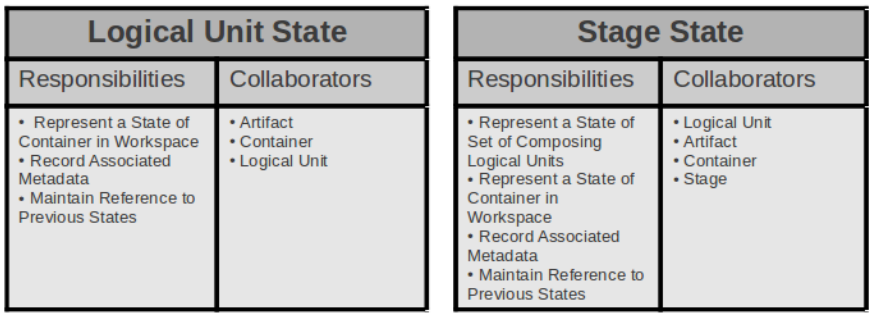


Figure 7.16: Logical Unit and Stage State CRC Cards

Role Relationships

In this section the relationships between the logical unit, stage and their respective states will be considered. The stage represents the overall project and is composed of an arbitrary number of logical units. Thus a stage is a singular instance within the system and that holds a one-to-many relationship logical unit.

The relationship between the logical unit and the stage exhibits characteristics of a part-whole compositional relationship, as the logical units are derived from the division of the overall project. Thus, the applicability of the Composite design pattern must be considered. The Composite design pattern's intent is to "[*c*]*ompose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly*" [GHJV95]. This pattern is applicable if a client handles a stage and logical unit uniformly. However, this is not the case. Each has a very distinct purpose that cannot be abstracted to provide the necessary uniformity. The primary purpose of the stage is to define and manage a set of logical units. It would be reasonable to allow a logical unit to define and manage a set of logical units, but the increased system complexity would reduce its usability. Therefore, the Composite pattern was not applied.

The history of either a stage or logical unit is represented as a set of states. To represent a relationship with a set of states, there must be a one-to-many relationship maintained between a stage or logical unit and their respective states. While it is true that the history of each is described through an arbitrary number of states, the one-to-many relationship may be reduced to a one-to-one relationship with the most recent state committed, i.e. the head state, being maintained. This is possible because the history is actually a sequence of states and each state maintains references to their previous states; which allows the entire history sequence to be traversed from a single

head state. Additionally, a reference to the most recent state committed or reverted to, i.e. the current active state, must also be maintained to support reseting the workspace.

Figure 7.17 depicts the relationships between the various roles resulting from these considerations. As seen, each role is related to a container and the relationships are generally mirrored between the stage and logical unit. The difference lies in that both the stage and its state maintain a one-to-many relationship to their respective counterpart, i.e. the logical unit and logical unit state.

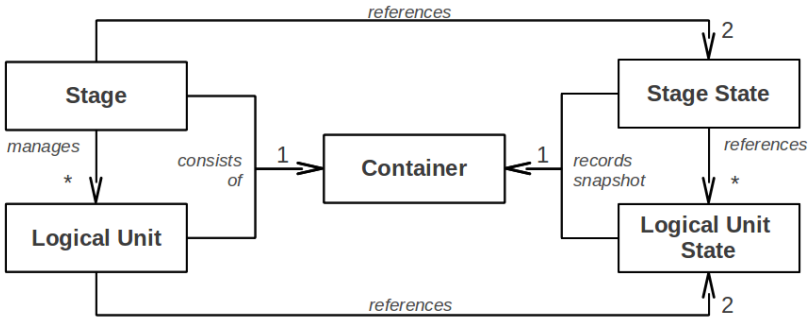


Figure 7.17: Logical Unit and Stage Role Relationships

Class Diagram Extension

In this section the new roles will be integrated into the class diagram of the versioning core described in Section 7.1.1: *Class Definitions* on page 69. This creates a complete representation of the internal structure and behaviors of the system.

When considering the previous roles and relationships, the mirrored structure surrounding the stage and logical unit indicate a symmetry that may be abstracted. Both elements may be committed, reverted, and maintain references to two states and a container. This common behavior can be abstracted into a superclass Committable.

The stage and logical units situation reflects the active versioning model and their references within the model must be persisted. To accomplish this, each element must have a means of storing and loading its respective referential information. Thus each must have a corresponding storage location within the repository. This presents a commonality shared by all elements of the versioning model; they are all persisted within the repository. This common behavior can be abstracted into a superclass PersistedElement, from which each element inherits the ability to persist its information in the repository.

The persisted elements may be further classified into one of two categories: *retrievable* or *committable*. Retrievable elements are responsible for retrieving information from the repository and depositing it into the workspace. It replaces the abstract Element class as the superclass of the Artifact, Container and State classes. Committable elements are responsible for signaling the user's intent to persist or change the current state of the workspace. The Stage and LogicalUnit are Committable elements. The class diagram resulting from the introduction of the stage and logical units is depicted in Figure 7.18 on the facing page.

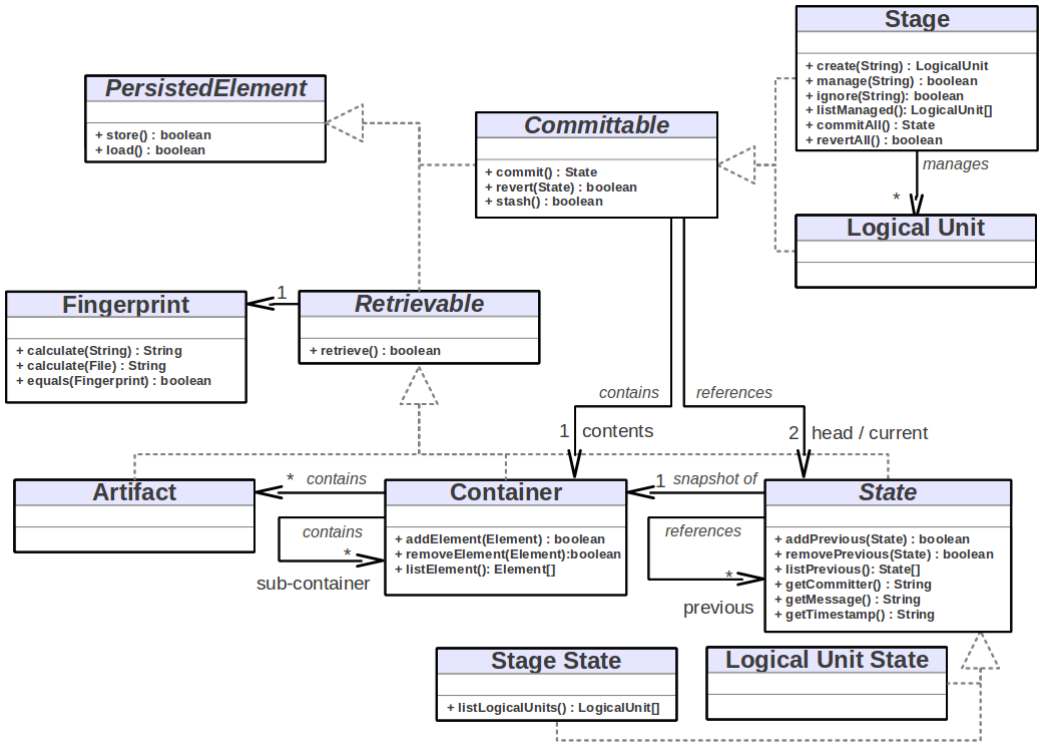


Figure 7.18: Final Versioning Core Class Diagram

7.2.2 Repository Design

In this section the repositories storage concept will be extended to support the persistence of the stage and logical units. First, the formatting for the committable element's information will be considered. Then, the designs for the retrievable and committable elements will be integrated to produce the overall storage concept.

Committable Element Formatting

This section considers how the relevant information of the committable element's and their respective states may be formatted and stored within the repository. First, the logical unit and the logical unit state will be considered and then the stage and its state will be considered.

The logical unit is responsible for maintaining a set of artifacts, organized within a single root container as well the references for its current situation, i.e. the head and current states. Each of these are fingerprinted elements and a fingerprint reference may be maintained. The abbreviation *HD* indicates that the referenced logical unit state is the most recent state committed or "Head" state. The abbreviation *CU* stands for "Current" and refers to the most recent state the logical unit either committed or reverted to. The abbreviation *ST* stands for "Stash" and refers to the active container's fingerprint. This allows an altered content, i.e. artifacts added, removed or changed, to

be persisted without explicitly committing the logical unit.

There is no change to the logical unit state's responsibility. It records a reference to the root container (CO), all previous states (PS) and the metadata (MD). The logical unit and its state formats are depicted in Figure 7.19.

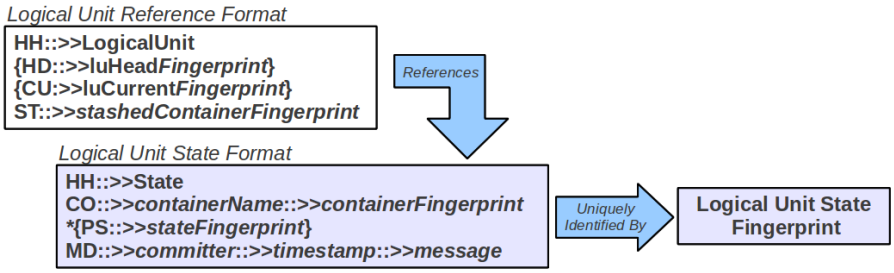


Figure 7.19: Persisted Logical Unit and State Format for Calculating Fingerprint

The responsibility of the stage is similar to that of the logical unit, but must also maintain references to each managed logical unit. Likewise, the stage's state format is similar to the logical unit's state format, but requires a means for maintaining the states of the managed Logical units. Figure 7.20 depicts the format for the stage and the stage's state. All previously identified abbreviations remain the same. The abbreviation LU is the only new abbreviation introduced and refers to a "Logical Unit". The stage's reference file maintains only the logical unit's name; which may be used to find and load the information recorded in the logical unit's reference file. Within the stage's state, the logical unit's information also includes the specific logical unit state associated with the stage's state.

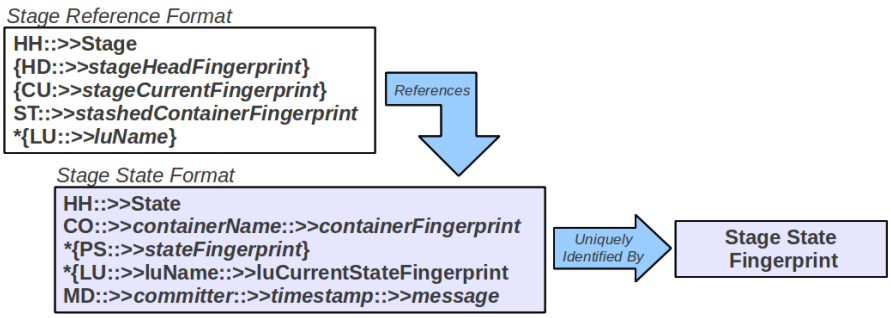


Figure 7.20: Persisted Stage and State Format for Calculating Fingerprint

Repository Organization

This section integrates the storage concepts for the committable and retrievable elements into a overall storage concept that describes the repository's organization. Unlike the retrievable elements, which employ a content addressable technique, the committable

elements maintain a static reference to a singularly persisted instance of their information. This means that the information must be persisted in a static location and the persisting of any updated or changed information overwrites the previously persisted information. Therefore, the repository must be split into two separate sections. One portion, to be named *fpStore*, maintains the retrievable element's information which is stored according to their fingerprint. The other portion, to be named *refStore*, maintains references that are used for the committable Elements, which are stored according to their name. Both folders will be encapsulated within the repository's root folder, to be named *.hydra*.

For example, given a stage with three logical units, *luA*, *luB* and mymathluC, the repository would have the structure depicted in Figure 7.21. References to each of the committable elements are maintained in the *refStore* subfolder. All content persisted by the retrievable elements are found in the *fpStore* subfolder and are accessed through their fingerprint.

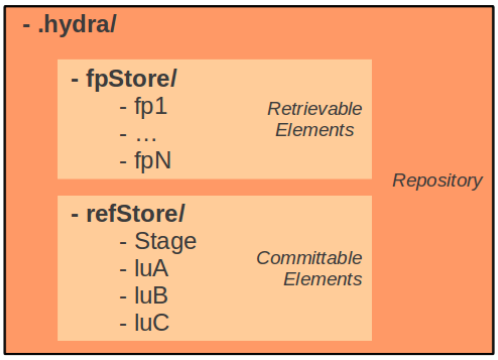


Figure 7.21: Repository Structure Example

7.2.3 Multi-Headed Integration Commits

The new versioning granularity level, the logical unit, affects the concept of a commit. In contemporary VCSs the entirety of the project, i.e. all artifacts and their interdependencies, is encapsulated in a single commit. The logical unit enables the each subproject to be managed autonomously and their committed states maintain the subproject artifact dependencies. The stage performs the function of subproject integration and maintains the subproject dependencies.

In this section, an example will be presented to depict the different aspects of the stage's committing functionality. The stage's characteristics introduce two new types of commit: (1) Integration and (2) Recursive Integration Commits. The integration commit simply records the current state of each logical unit. The recursive integration commit first forces each logical unit to commit any changes and then records the current state of each logical unit.

Commit One (Integration) – Most Recent State Integration

Consider a project consisting of three logical units, *luA*, *luB* and *luC*. Each has been defined and has made a single initial commit. After testing the integration of the three states, it is determined that they represent a coherent overall project state. In order to record this, the stage creates an integration commit. The integration commit simply records each of the current states of the logical unit. Figure 7.22 is a depiction of this initial integration commit. Each of the logical units, listed along the left side of the diagram, have created a single state, depicted as a light blue circle and annotated with the logical unit's letter and the number of the commit. The dark magenta state represents the stage's state, which reflects the integration of the *A*.1, *B*.1 and *C*.1 logical unit states.

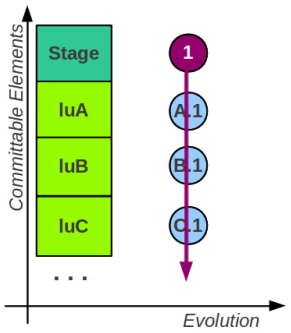


Figure 7.22: Initial Integration Commit

Commit Two (Integration) – Selective State Integration

Now consider the situation where the logical units *luB* and *luC* have been further developed and each creates a new logical unit state, *B*.2 and *C*.2. No further changes are introduced to logical unit *luA*, thus no new logical unit state is created. After testing the integrated system, it is determined that the overall system performs better with the first state of logical unit *luC*. This could happen when the new version of *luC* introduces new functionality that is not yet used within the system. The new state represents a forward step for the independent logical unit, but results in a degradation of the overall system. Thus, a new system integration commit should reflect the collection of states: *A*.1, *B*.2 and *C*.2.

The resulting situation is depicted in Figure 7.23 on the next page. The new stage state is annotated as number two and traces a path connecting the three integrated logical unit states as described. The gray States are those that were not created during this step. As can be seen, the stage is not limited to only the most recent state of a logical unit. It is able to combine any set of logical unit states to create a coherent overall state.

This example represents the independent management of each subproject as well as the overall project, which is not supported by contemporary VCSs. Within contemporary

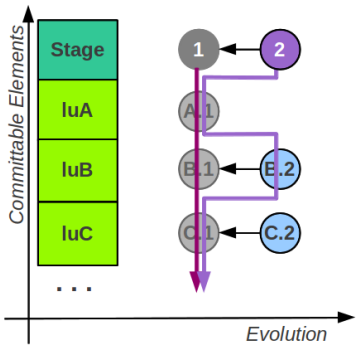


Figure 7.23: Second Integration Commit

VCSs, this integration step is implicitly part of the singular commit and thus could not be represented.

Commit Three (Recursive Integration) – Most Recent Change Integration

Next consider the situation where alterations have been made to each of the logical units, *luA*, *luB* and *luC*, but the changes have not been committed. The current resulting overall project is tested and the combination of all changes represents a coherent system state. At this point, the changes of each logical unit have not been committed and as such the stage's standard integration commit is not sufficient to record the new state as it can only reference persisted logical unit states.

The situation can be dealt with by first committing each logical unit and then creating a new stage state that references the new commits. This process could be executed manually or automatically, depending on the user's. The recursive integration commit, is introduced to automatically execute this sequence of steps. This automatic recursive integration commit, is the only type of commit available within contemporary VCSs.

Figure 7.24 on the following page depicts the described situation. The purple stage state annotated as number three represents the recursive integration commit. This first creates the new logical unit states, $A.2$, $B.3$ and $C.3$, which are represented as dashed annotated circles. The dashed lines reflect that the commit was automatically created by the system.

Commit Four (Recursive Integration) – Selective Change Integration

No consider the following situation. Logical unit *luA* has been altered and committed, creating logical unit state $A.3$, and then further altered but not committed. Logical unit *luB* has been altered but not committed. Logical unit *luC* has been committed, creating $C.4$. Upon testing the system, it is determined that the old logical unit state $C.3$ integrates better with the overall system then the new state $C.4$. A recursive integration commit would first automatically create of the logical unit states $A.4$ and $B.4$. No new state would be created for logical unit *luC*, since there is no changes reflected in the

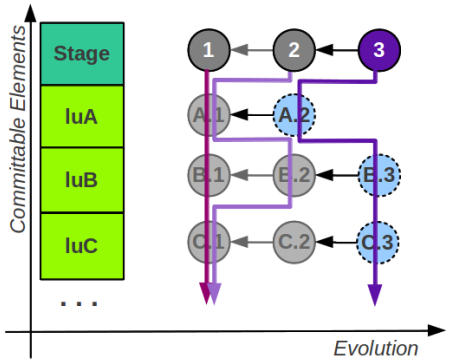


Figure 7.24: First Recursive Integration Commit

workspace. Next, it would commit the system's state which would reflect the integration of the following logical unit states: $A.4$, $B.4$ and $C.3$.

Figure 7.25 depicts the resulting recursive integration commit. From the diagram it can be seen that even with a recursive commit, the user has the capability of selecting any commit, if the respective Logical Unit has not been altered.

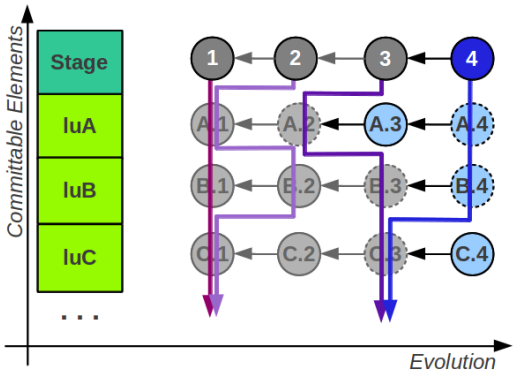


Figure 7.25: Second Recursive Integration Commit

Impact on VCS Capabilities and Workflow

As described in the previous examples, the introduction of the logical unit as a new level of versioning granularity has a significant impact on the capabilities and workflow of a VCS. It provides the capability to support the independent evolution of each logical unit. As well as an improved ability to integrate the system, by providing a wider selection of states to choose from. Finally, this new functionality comes with an increase in the system's management complexity. However, the entirety of the complexity could be ignored and the system could be used exactly like a contemporary VCS through the use of the recursive integration commit supported by the system's stage.

7.3 Validity Tracking

This section investigates how the concept of validity tracking may be integrated into the versioning core. First, the property based and path based methods will be reviewed and compared. Next, these concepts will be integrated into the versioning core through the extension of the state element. Finally, a set of examples will be presented to reinforce the intended purpose.

7.3.1 Property vs. Path Based Validity

As described in Section 1.1.4: *Validity and Valid Path Evolution* on page 6, the validity of a state may be determined based on two different but orthogonal means. It may be determined either based on a property or set of properties associated with the state or it may be determined dynamically from its historical context.

Each state has a set of properties associated with it. These properties are typically considered its metadata and consist of at least: the party responsible for creating the state, the reason the state was created and the time at which it was created. In order to support property based validity, a property specifically defining the state's validity must be also be associated with each state. Assume that each state has a property, *isValid*, associated with it that defines its validity. This type of validity derivation assumes that the validity of a state can be directly derived from its characteristics and must be explicitly set. Figure 7.26 depicts the employment of this technique.

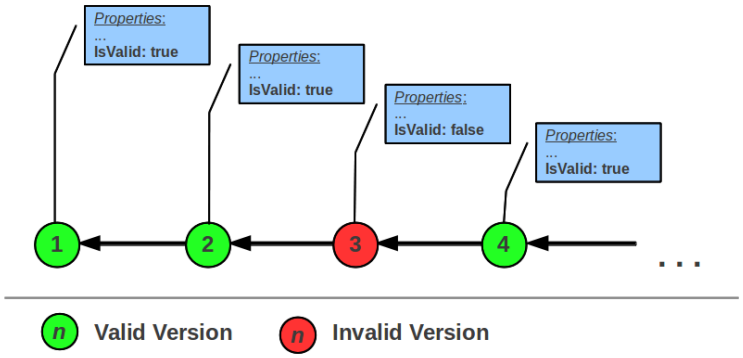


Figure 7.26: Property Based State Validity

Path based validity extends the inter-state references instead of the properties in order to define validity. Each state maintains not only the standard previous state, which is occurrence based, but also a reference to its valid predecessor. Through the referential maintenance, validity becomes a property of a state's historical context not a property of its internal characteristics. This allows validity to be determined based how an artifact was altered, for example concurrently, or from which state a state is conceptually derived. While this technique is more complex, it provides a number of advantages over property based validity. Path based validity:

- Removes the need to explicitly set each state's validity.
- Is not restricted to system path, it explicitly describes a sequence of valid states.
- Can describe a single valid history distributed across several branches.

Figure 7.27 depicts a path based validity scenario distributed across multiple branches and dynamically describes the validity of a state based on the defined valid path references.

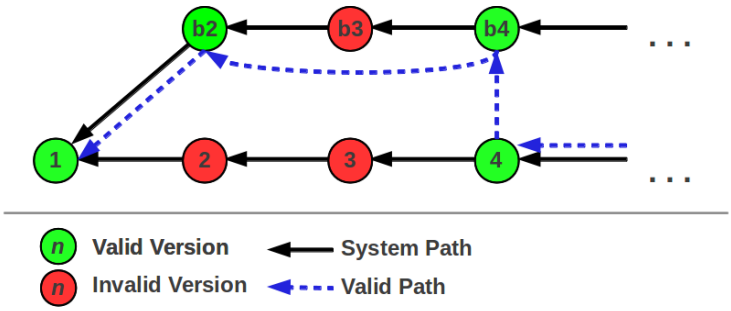


Figure 7.27: Path Based State Validity

The property based technique is relatively simple to implement and understand. It is adequate for supporting the validity functionality needed in most systems. However, the overall validity context is not represented. The path based technique provides a means to dynamically describe inter-branching historical validity. The valid path describes the valid history of an artifact while implicitly ignoring any invalid states. Finally, both of these two methods of defining validity, content and context based, are orthogonal as they deal with differing aspects of validity. One deals with the validity tied to its content and the other deals with its validity when considering the overall historical context.

7.3.2 State Validity Extension

In this section, the extension of the state to support the described validity concepts will be introduced. First property based and then path based validity will be integrated.

The integration of the property based validity requires updating the state class and its persistence format. A new boolean field, isValid, and methods to set and query the field must be introduced to the class. Additionally, the field must be represented within the state format to persist its value with the other associated metadata.

The integration of the path based validity has very similar needs. A new state field, validPrevious, maintaining a reference to the previous valid state is required. Additionally, appropriate setting and querying methods are need. Finally, the state's reference to its valid previous state must be represented in state's format and persisted with the state's associated metadata.

Figure 7.28 on the facing page depicts the updated state class and format. The bold-faced text indicated the additional information required to support both forms of validity.

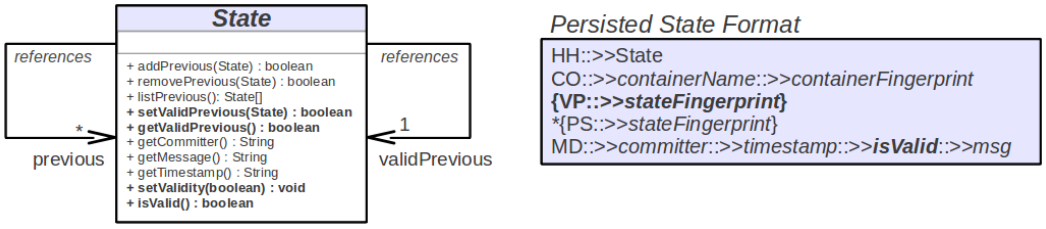


Figure 7.28: State Validity Support Extension

7.4 History Manipulation

In this section ways to improve the flexibility of the system with respect to the manipulation of history are considered. A history can be changed in two ways: (1) introduction of a new state or (2) alteration of a previously introduced state. Means for increasing the system's flexibility in both areas will be considered. Inserting a state at an arbitrary location and defining a temporary state are two ways in which this system increases its flexibility while introducing new states. Adjusting the definition and calculation of a state's fingerprint enables the alteration of state's content and metadata without affecting its histories referential integrity, this improves the system's ability to alter previously persisted states.

7.4.1 Insert and Temporary Commits

Traditionally, new states are only introduced as a new head state in a history. However, the means to support the insertion of a new state at an arbitrary location within a history is already supported by the versioning core. In order to insert a new state, four steps must be followed: (1) identify the new states predecessor and successor, (2) commit new state, (3) set new states previous state to the predecessor and (4) add the new state as a previous state of the successor.

It is assumed by contemporary VCSs that the content to be represented by a state is always available. However, as may occur within α-Flow, a state may be realized within the system before the necessary information that it describes being present. This introduces the need to create a temporary commit. A temporary commit is a commit without defined content. The content is expected to be provided later.

7.4.2 Fingerprint Addressable Storage

The information represented by a state is persisted in a location defined by its fingerprint. This concept is drawn from the content addressable storage paradigm. It is critical that each unique state has its own unique fingerprint. However, the unique fingerprint must not necessarily include all information or the entire content of the state. Reducing the amount of information from which the fingerprint is calculated, increases the amount of information that may be altered without changing the states fingerprint. For example if the state's associated purpose is not included in the calculation of the fingerprint. Then, the state's purpose may be altered without impacting storage location.

A history depends on the fingerprints to maintain the inter-state references. As long as the fingerprints are not altered, then the referential integrity of the history is not harmed. In order to create complete flexibility while still guaranteeing the uniqueness of the fingerprint, the fingerprint may be generated based on a Universal Unique Identifier (UUID) and not the state's content or metadata. Figure 7.29 depicts the altered persisted state format. The area highlighted in a light gray blue indicates the information used to calculate the state's fingerprint.

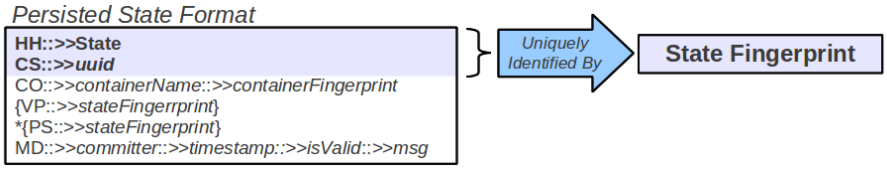


Figure 7.29: Fingerprint Addressable Storage State Format

7.5 Summary

This chapter focused on the design of the internal versioning core that represents the heart of the Hydra mVCS. First, the fundamental elements of the system, i.e. the artifact, container and state, were introduced and described. These elements are responsible for representing differing aspects of a workspace and are capable of persisting and returning the workspace to a given state. Effectively supporting the commit and revert functionality. Committable elements, logical unit and stage, capable of managing the system and coordinating the activities of the basic retrievable elements were then introduced and described. The logical unit introduces a new versioning granularity level and supports the independent management of subprojects. The stage is responsible for representing the integration of the various subprojects into a cohesive overall project.

Next, the concepts of validity tracking were introduced. Validity can be derived either based on the inherent characteristics of a state or its context within a history. Property-based and Path-based means of deriving a state's validity were compared and their concepts were integrated into the versioning core's design.

Finally, the concept of fingerprint addressable storage was derived from the content addressable storage paradigm. In this paradigm, the storage location is not necessarily defined based on the complete content; rather it is derived from a uniquely identifying fingerprint. This allows the content and metadata associated with a state to be altered without changing the fingerprint and thus disrupting the integrity of the overall system's state references.

8 Design – User Interfaces and Subsystems

In this chapter, the design of the User Interface (UI) enabling interaction with a user and the various subsystems supporting the versioning core are introduced. First, the UI's design will be introduced focusing of the major components of interaction. Next, the persistence, logging and differential calculation subsystems will be described.

8.1 User Interface

The versioning core design in the previous chapter provides support for embedding Hydra mVCS within other applications. However, most often VCSs are used as a standalone program or is integrated into the developmental environment. In order to support these employments, appropriate UIs are necessary.

First, the functionality expected to be supported within the VCS was considered. Once the system's functionality was defined, it was encapsulated within a *command*. The command pattern [GHJV95] provides a means for encapsulating the various functions and make them available as executable object. Then support for the user's access to this functionality was designed.

The user will interact with a UI, to request the available functionality. The UI is responsible for identifying the associated command and information necessary for its execution. The UI will the requests the execution of the functionality from the command. The command in turn coordinates the execution, which is realized by the previously described versioning core. Figure 8.1 depicts this interaction as an activity diagram and highlights the area that will be designed in this section.

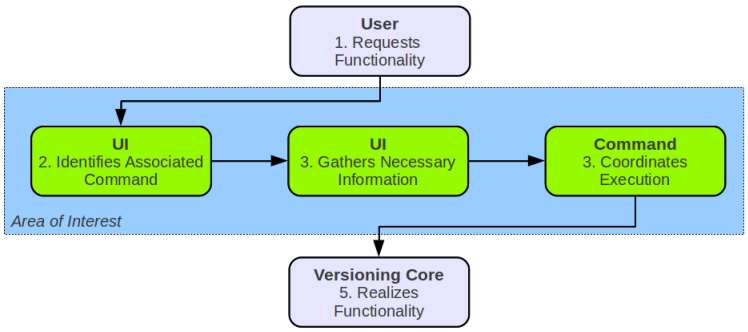


Figure 8.1: User Interface Activity Diagram and Area of Interest

8.1.1 Commands

In this section the functionality supported by the UIs will be considered and then will be organized into a set of encapsulating commands. This set of functionality encompasses and extends the standard expected functionality associated with VCSs.

Identifying Functionality

Much of the functionality supported by this system can be derived from other VCSs, as they all provide functionality to accomplish similar tasks. Additionally, the employment of common terminology and functionality will simplify the transition of a user between systems. Common functionality associated with most contemporary VCSs includes:

- Add file to versioning
- Remove file from versioning
- Move/Rename file*
- Commit state of workspace
- Revert state of workspace to previously committed version
- Log or describe history of changes
- Determine status of the workspace (i.e. which files have been changed)
- Describe differences between versions of files*
- Create new branch*
- Change branch*
- Merge changes from another branch*

As described in this projects motivation, this system does not implement all of the behavior commonly associated with a VCS. The most notable capabilities not supported are branching and merging and are annotated with an asterisk in the listing. A future version will implement these capabilities but where not deemed necessary for the successful implementation of this version. While some common behavior is not supported, other new concepts are introduced.

Extending Functionality

The logical unit has the greatest impact on the system's capabilities as it introduces multiple autonomous actors within the system. Each maintains a set of files, which may be autonomously committed or reverted, and manages its own independent history. Therefore, it becomes necessary to identify the actor when adding or removing a file, committing or reverting the workspace state and logging a history.

The stage not only provides capabilities similar to the logical unit but adds a set of new managerial capabilities. It must be able to create new logical units, manage or ignore logical units and integrate the states of the managed logical units into a single coherent system state.

Table 8.1 on the next page depicts the extension of the common functionality associated with contemporary VCSs that is necessary to employ the extended capabilities of this system.

Capability	Common	Extension Required
Add File	Yes	Identify Logical Unit or Stage
Remove File	Yes	Identify Logical Unit or Stage
Workspace Status	Yes	Identify Logical Unit or Stage
Commit Workspace	Yes	Identify Logical Unit or Stage
Revert Workspace	Yes	Identify Logical Unit or Stage
Log History	Yes	Identify Logical Unit or Stage
Create Logical Unit	No	
Manage Logical Unit	No	
Ignore Logical Unit	No	

Table 8.1: VCS Capabilities and Necessary Extensions

Organizing Commands

Once the set of capabilities has been identified, they must be organized into a manner which is simple to understand and implement. The majority of commands are applicable for both logical units and the stage and this introduces divergence along two axis: task and actor. The most important design decision is to determine which axis takes precedence and results in the best design. However, when taking the additional commands into account, it was determined that the commands be first split along the actor. This is because there is a set of tasks that are only applicable for the stage.

Figure 8.2 on the following page depicts the structural decomposition of commands supporting this system. The commands are grouped according to the actor on which they operate. CommandStage is an abstract representation of the commands employing the stage and CommandLogicalUnit represents the abstract command employing a logical unit.

Implementation Concerns

This decomposition is the extent of the design that can be considered at this point. The intent of each Command class is defined and its implementation can be derived from the capabilities provided by the versioning core. Each of the Command classes will inherit and implement a singular method, #execute(), that is responsible for executing the desired functionality. This allows all Commands to be handled uniformly within the system.

8.1.2 Command Line Interface

In this section, the means of interacting with the user over a Command Line Interface (CLI) will be considered. The CLI is responsible for (1) receiving instructions from the user, (2) organizing necessary information, (3) requesting the execution of the appropriate command and (4) displaying the result to the user.

The first and fourth steps are relatively simple. The first step requires the prompting

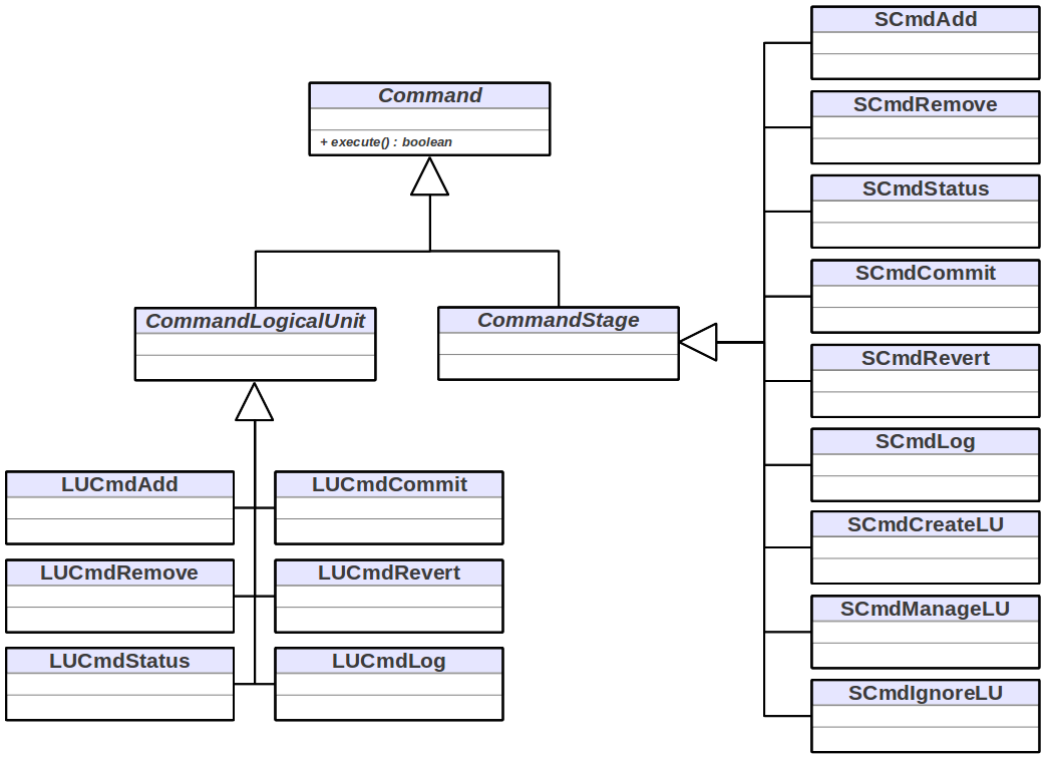


Figure 8.2: Command Class Diagram

for and accepting a user's request over the command line terminal. The fourth step consists of consolidating the results from the command and displaying it on the command line terminal. Steps two and three are significantly more complex and require the parsing of the user's request and determining what information is necessary.

Traditional Command Line Parsing

Parsing of the command line is a common task within software development and there are a number of open source implementations available for usage. One example is Apache's Common CLI[1]. However, it, like others, supports a broad range of functionality, most of which is not necessary. Additionally, it separates the parsing from the object that has explicit knowledge over what is needed. A separate parsing unit requires the explicit knowledge of each command's parameters and couples tightly to each command.

Therefore, it was decided to develop a lightweight means of parsing that is directly integrated into the command. This allows a new command to be integrated into the system with no adjustments necessary in a separate parsing unit. All information pertaining to the specified functionality is encapsulated within the command.

[1]Apache Commons CLI Homepage: `http://commons.apache.org/cli`

Regular Expression Command Line Parsing

Regular expressions provide a common abstraction used for the parsing and manipulation of strings. They provide the means to define an acceptable input string and access the various defined elements within the string. If each command is capable of recognizing and parsing an appropriate inputed command line, then the UI may then request this functionality directly from the command. This trades the conditional operations within the parsing unit for a loop in the UI over the defined commands. An activity diagram depicting the control flow of the CLI user interaction is presented in Figure 8.3.

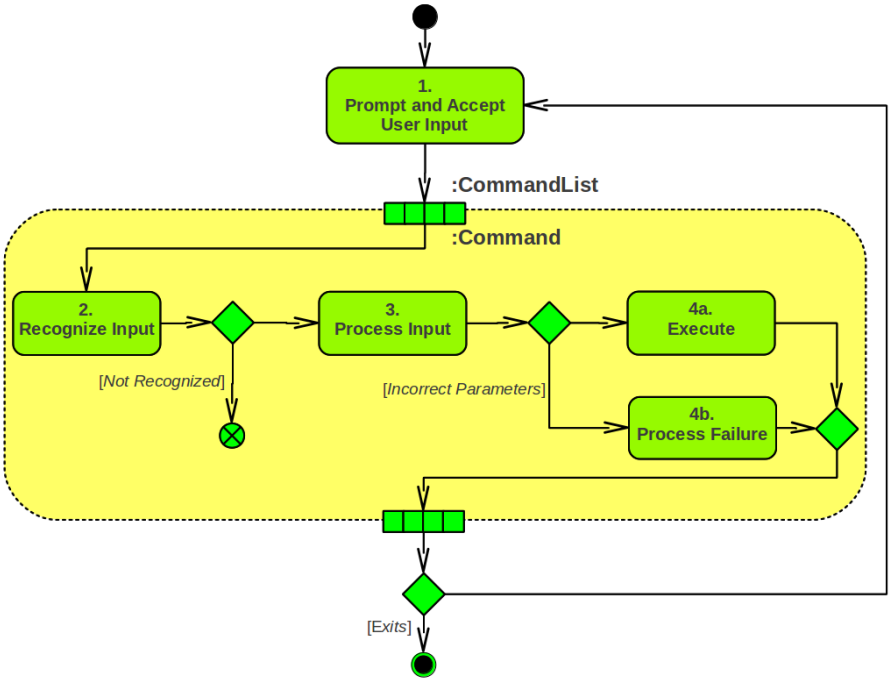


Figure 8.3: CLI Command Processing Activity Diagram

As depicted, each command follows the same general algorithm during the processing of the command line. First, the command line input is tested to determine if it is recognized (2). Then, if the command line is recognized, the parameters are processed (3). If the parameter processing is successful (i.e. the correct parameters are included in the command line input), then the desired behavior is executed (4a). Otherwise the user is notified of incorrect parameters (4b). Finally, the command is queried to determine if the system should exit.

Template Method Command Processing

The vast majority of this algorithm can be encapsulated within a template method. The intent of a template method is to "*[d]efine the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure*" [GHJV95].

In order to support this abstraction a superclass, CommandRegex, is introduced. The superclass defines the previously described processing algorithm and the subclasses define their respective regular expressions, extract matched parameters and execute the desired behavior. Figure 8.4 depicts the resulting class diagram, the individual Command classes are left out for simplicity's sake.

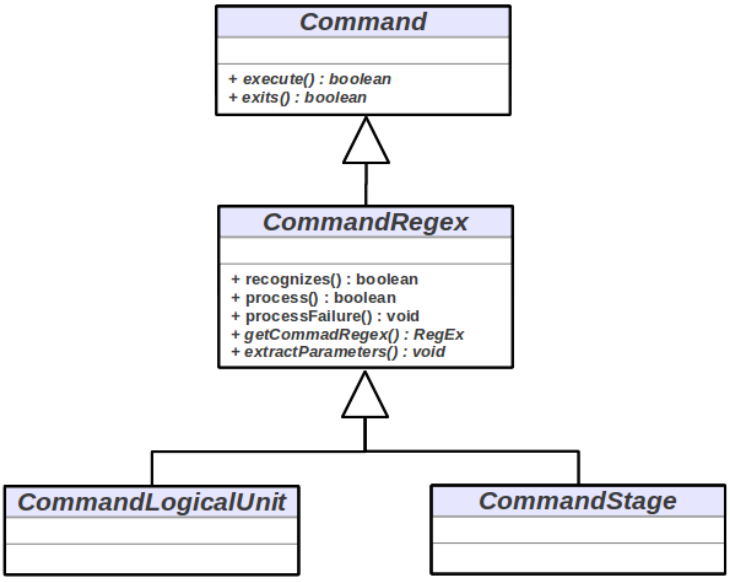


Figure 8.4: Command Class Diagram Including CommandRegex

Handling User Input

Once all the functionality is encapsulated within the Command class, the next step is to define a class that prompts for and gets the user's input. Once it has the user's input, it loops though its repertoire of commands and asks each if it recognizes the input. Once a command has been found that recognizes the input, the command line is processed and the command is executed. Finally, the CLI may be executed in single command or interactive modus. In interactive modus, the CLI continues to prompt for user input until the system is explicitly exited. In single command modus, the desired command is executed and the system exits.

Command Formats

One final consideration is the format of the commands the user will input to request the execution of functionality. Table 8.2 on the next page depicts a listing of these VCS capabilities and their command line usage. The bold-faced words must be explicitly written (case insensitive) and the italicized words indicate information that must be inserted by the user. Commands operating on a designated logical unit are prefixed with

the letters *'lu'* and must explicitly identify the logical unit upon which they operate. Commands operating on the stage are prefixed with the letter *'s'*.

Capability	Logical Unit	Stage
Add File	**luAdd** *luName* **-e** *eName*	**sAdd -e** *eName*
Remove File	**luRemove** *luName* **-e** *eName*	**sRemove -e** *eName*
Workspace Status	**luStatus** *luName*	**sStatus**
Commit Workspace	**luCommit** *luName* **-m** *message*	**sCommit** {**-r**} **-m** *message*
Revert Workspace	**luRevert** *luName stateFingerprint*	**sRevert** *stateFingerprint*
Log History	**luLog** *luName*	**sLog**
Create Logical Unit	N/A	**sCreate** *luName*
Manage Logical Unit	N/A	**sManage** *luName*
Ignore Logical Unit	N/A	**sIgnore** *luName*

Table 8.2: CLI Command Usage

8.1.3 Graphical User Interface

In this section the design of a Graphical User Interface (GUI) will be introduced. The GUI should provide the same functionality as the CLI but also provide a visual depiction of information and a point and click or menu based modus of interaction with the user.

Component Parts

The GUI is composed of three parts: (1) explorer, (2) menu and (3) console. The explorer provides the visualization of the system. It is further divided into two parts: a (1) selectable committable elements (i.e. logical unit or stage) portion where a user may select which element will be considered and a (2) visualization part which presents the history, status or contents of the selected element. The menu provides the user with the ability to select a desired action to be executed. The console is an embedded CLI that allows the user to interact with the system over the command line if desired. The top portion accepts the input from the user and the bottom portion is a scrollable text field that displays the same textural output as the CLI. Figure 8.5 on the following page is a depiction of the described GUI layout.

Class Diagram

An initial class diagram can be developed which describes the GUI's structural organization. Figure 8.6 on page 101 is a depiction of the resulting class diagram. The commonalities, namely the command list and the interactive behavior, of the GUI and CLI classes have been abstracted to a super class UI. This design could be further specified, but then it would be to constrictive during the implementation. Therefore, it

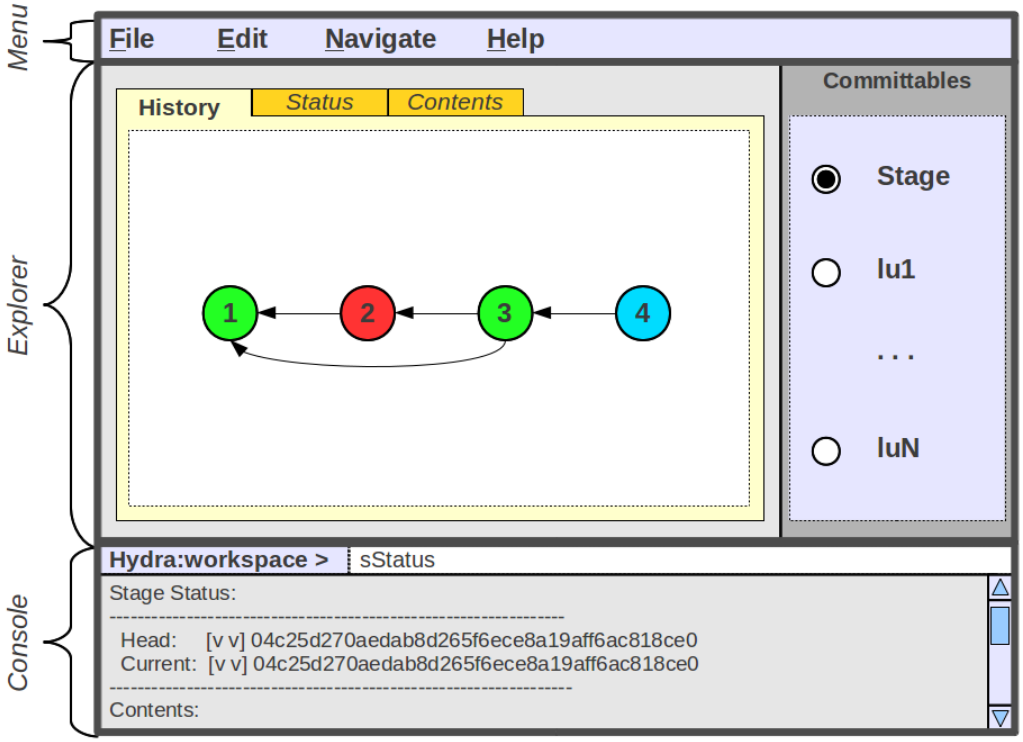


Figure 8.5: GUI Layout Design

remains vague but defines the critical components and the intended overall structure of the system.

8.2 Persistency Subsystem

In this section the persistence subsystem will be designed. The purpose of the persistence subsystem is to encapsulate the functionality needed to persist the state of the versioning model elements so they may be be restored at a later time.

First, the key components and definitions of the terms used will be covered. Next, the necessary functionality will be designed and encapsulated into functional units, i.e. Data Access Objects (DAOs), that are responsible for providing the desired capabilities. Finally, differing storage strategies will be introduced to provide a means to configure the system to the demands of its employment.

8.2.1 Components

The persistence subsystem is responsible for transferring information between three locations: (1) workspace, (2) repository and (3) versioning model. The workspace represents the portion of the system that is visible and editable by a user. The repository is responsible for maintaining the data after the application exists. The versioning model

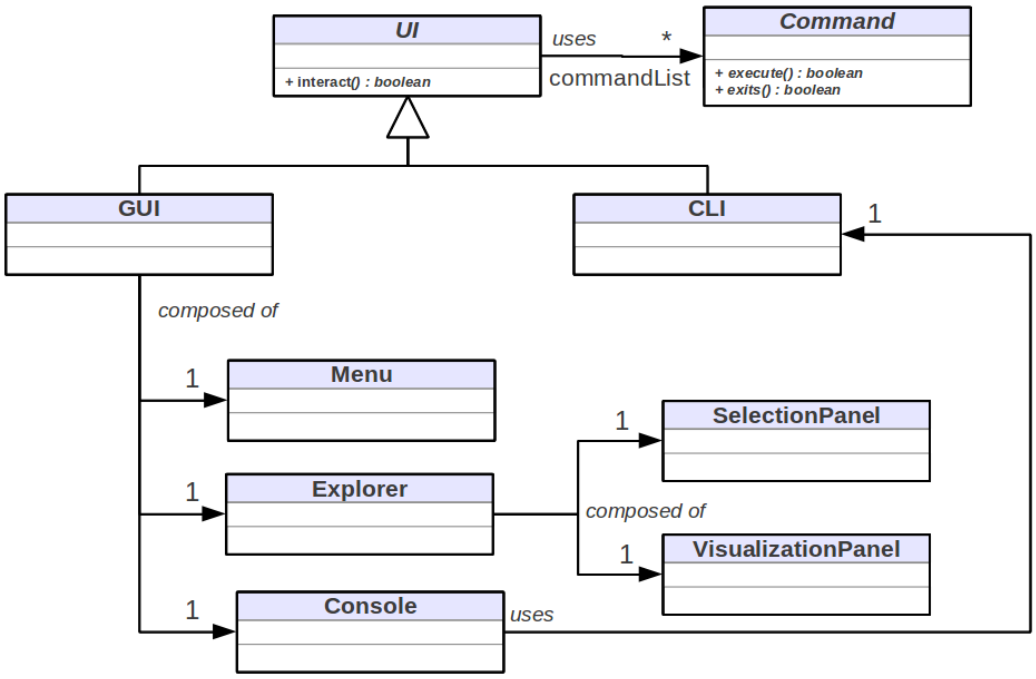


Figure 8.6: User Interface Class Diagram

provides a working memory representation of the information manipulated by the VCS.

The workspace only contains and thus deal with artifacts and containers. The repository and versioning model deal with all persisted element types (i.e. artifacts, containers, states, logical units and the stage).

8.2.2 Terms

The exchange of information between the various locations requires a clarification of terms to avoid confusion when discussing the system's actions. A summary of the terms are as follows.

Add – Introduces a new element into the versioning model based on the information present within the workspace.

Refresh – Updates modeled element's information based on its current state within the workspace.

Store – Persist the current state of a file or folder within the workspace in the repository.

Retrieve – Return a file or folder in the workspace to a state previously persisted in the repository.

Load – Return a versioning model element to a state previously persisted in the repository.

Record – Persist the current state of a versioning model element in the repository.

Figure 8.7 provides a visual depiction of the subsystem's components and their interactions. Unlike the other interactions, the versioning model has no means of directly altering the workspace. Alterations to the workspace are accomplished by indirectly requesting a persisted state of an artifact or container to be retrieved from the repository by the representing versioning model element.

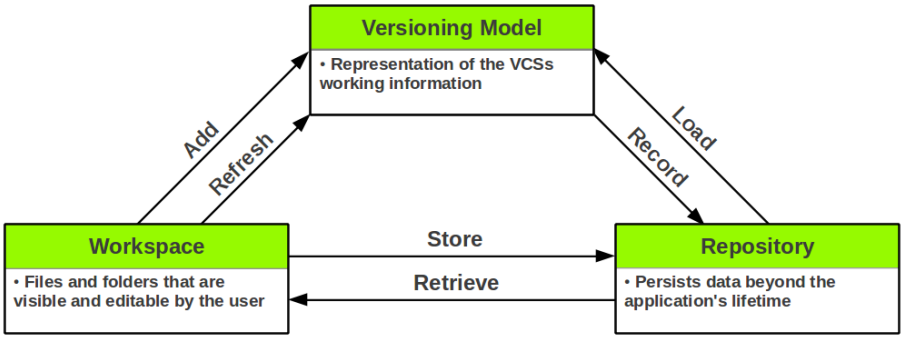


Figure 8.7: Component and Interaction Overview

8.2.3 Functionality

The persistence subsystem is responsible for transferring data between the repository and either of the other components (i.e. workspace or versioning model). To fulfill this responsibility it must provide support for the four specified operations: (1) store, (2) retrieve, (3) record and (4) load.

Store/Retrieve – File Content Transfer

The store and retrieve functionality deals with the transfer of file content between the workspace and the repository. Storing file content consists of two steps: (1) identify the repository storage location through the content's fingerprint and (2) transfer the file's content to the designated location. Retrieving the file content transfers the data in the opposite direction and also consists of two steps: (1) identify workspace location through the respective versioning model element's metadata and (2) transfer the file's content from the repository to the designated workspace location. Figure 8.8 on the next page provides a visual representation of this functionality.

Record/Load – Element Metadata Transfer

The record and load functionality deals with transferring an element's metadata state between the versioning model and repository. This functionality is similar to that described in the previous section, but the content to be transferred must be dynamically composed from the element's current metadata.

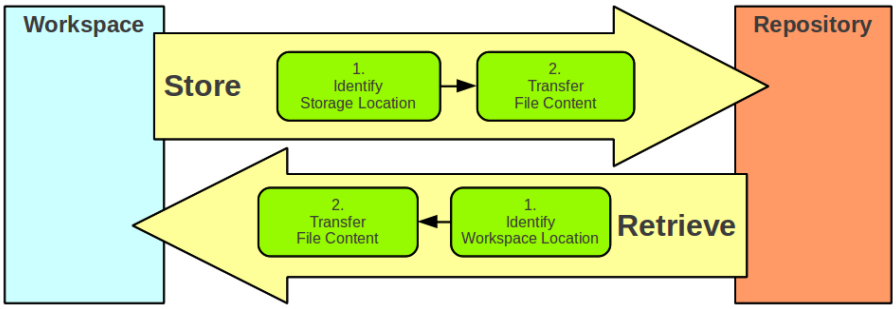


Figure 8.8: Store and Retrieve Functionality Visualization

To persist a model element's state it must first be transformed into a form appropriate for its persistence. This transformation is simply the organization of the element's metadata into the fingerprint formats described in Section 7.1.2: *Fingerprints – Uniquely Identifying Content* on page 73. Therefore, recording the state of a model element is accomplished through three steps: (1) transform the model element's metadata into a formatted fingerprint string, (2) identify the repository storage location through its fingerprint and (3) transfer the element's string representation to the designated storage location.

Loading a model element's state from the repository into the versioning model transfers the element's metadata in the opposite direction and instead of composing a string representation, the given string representation must be parsed to derive the contained metadata. The resulting process follows three steps: (1) identify the element's persisted location, (2) retrieve element's persisted string representation and (3) parse the string representation into assumed metadata form.

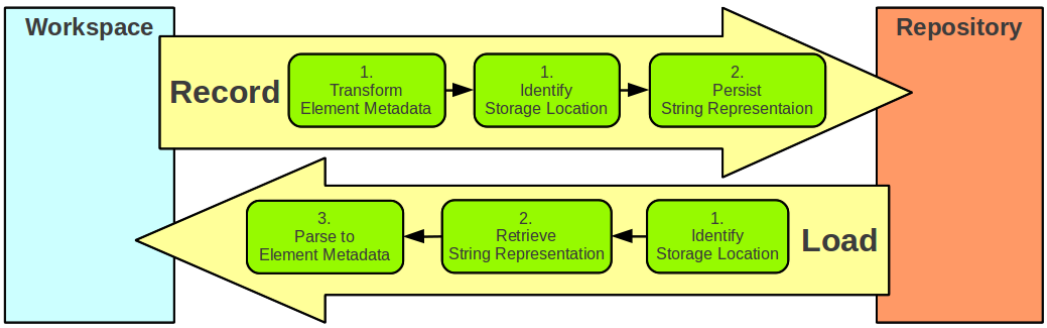


Figure 8.9: Record and Load Functionality Visualization

8.2.4 Data Access Objects

Data Access Object (DAO) are used *"to abstract and encapsulate all access to the data source. The DAO manages the connection with the data source to obtain and store data"* [CK03]. Employing the DAO design pattern decouples the versioning core from

the persistency subsystem and allows the underlying persistency infrastructure, e.g. file system or database, to vary without the need to alter the versioning core.

However, the basic DAO design pattern must be extended to meet the needs of the VCS. This is because it must support the transfer of data between all three of the previously described persistency subsystem components, instead of the generally assumed two locations. As described in Section 8.2.3: *Functionality* on page 102, the DAO must be capable of transferring data not only to the respective versioning model element, commonly referred to as the business object by the design pattern, but also between the data source, i.e. the repository, and the workspace.

The DataAccessObject is an interface that specifies four methods; one for each of the data transfer operations: (1) store, (2) retrieve, (3) record and (4) load. Figure 8.10 provides a visual depiction of the interface. The figure also depicts the separate implementations for each of the respective versioning model elements. These implementations encapsulate the respective element's data storage access and processing functionality needed to support each of these operations.

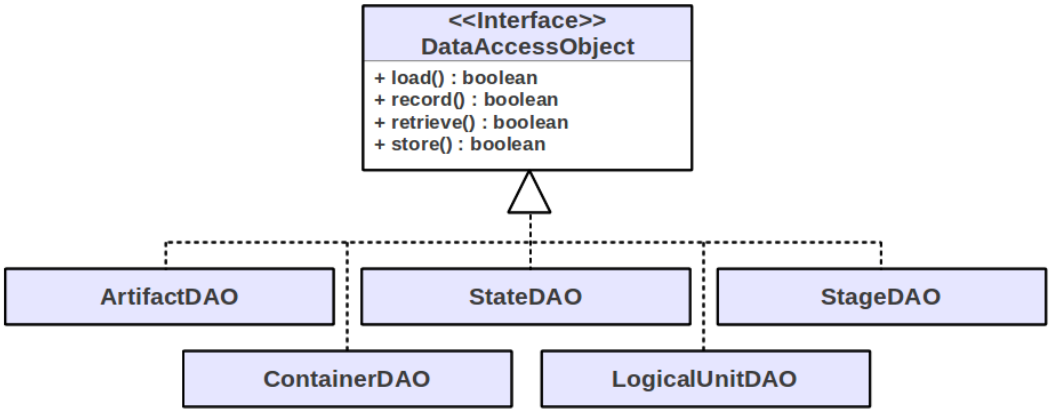


Figure 8.10: Data Access Object Class Diagram

8.2.5 Configuring the System to the User

The DAO pattern provides a decoupling of the versioning core from the underlying data source employed, but does not provide the means for altering the VCS's behavior to conform to the needs of the employing system. For example, α-Flow places an emphasis on the size of the repository and would benefit from the compression of the persisted data. The common software developer emphasizes the VCS's reaction speed and would benefit from the use of Java's New Input/Output (NIO) techniques.

When considering the activity diagrams presented in Section 8.2.3: *Functionality* on page 102, the Input/Output (I/O) functionality can be reduced to three operations: (1) transfer content from one location to another, (2) store a string in a specified location and (3) retrieve a string from a designated location. These operations may be extracted from

the DAO and encapsulated into a separate class that describes the specific algorithm or strategy that is used to transfer the data.

Storage Strategies

The strategy pattern's purpose is to "*define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it*" [GHJV95]. This pattern allows a number of different storage algorithms to be defined and the most appropriate to be selected, based on the employing system's needs. Thus, allowing the VCS to be configured to emphasize either reaction time or repository size.

In order to integrate this functionality into the system, the DataAccessObject interface must be refactored to an abstract class which maintains a reference to the StorageStrategy. The StorageStrategy in turn is an interface that has a number of concrete implementations; each emphasizing different performance aspects. Figure 8.11 provides a visual depiction of the resulting persistency subsystem's class diagram. Three storage strategies are defined. The NIOStorageStrategy employs Java's NIO functionality and emphasizes the system's reaction speed. The ZipStorageStrategy and GZipStorageStrategy both employ compression functionality found in the java.util.zip package and provide the means to compress the persisted data to reduce the size of the repository.

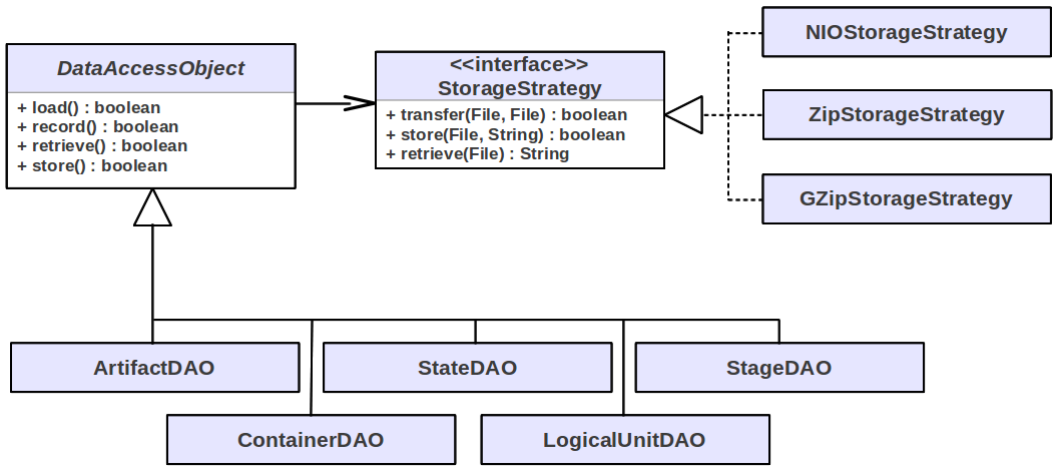


Figure 8.11: Persistency Subsystem Class Diagram

8.3 Logging Subsystem

The logger provides the system's basic logging capabilities. It is responsible for recording information within a log file as requested by the system according to differing levels of importance. While there are several standard logging solutions, for example Log4J[1],

[1]Log4J Homepage: `http://logging.apache.org/log4j`

they all implement functionality beyond the needs of the system and their implementation size is greater then the entire versioning system. As described in the CLI parsing consideration, it would be inappropriate if a utility apparatus represents a greater weight than the system which it is supporting. Thus a lightweight logging apparatus is introduced here.

8.3.1 Logging Levels

The logger recognizes six different logging levels: (1) debugging, (2) informational, (3) warning, (4) exceptional, (5) critical and (6) no logging. *Debugging* refers to logging entries that may assist the developer in debugging the system and analyzing its control flow. *Informational* refers to logging entries that may better help understand what the system did. *Warnings* refer to situations where some questionable behavior was observed by the system. *Exceptional* refers to the occurrence of a situation that should not occur. *Critical* refers to a situation in which a system failure occurs and the system must exit. *No Logging* identifies to the logger that is should ignore all logging requests. These logging levels are encapsulated within the enumeration LoggerLevel.

8.3.2 Logger Design

The logger itself then provides a singular point of contact which is responsible for collecting the logging requests from the various parts of the system into a single location. It offers the ability to log information at each of the previously described logging levels, ability to query the state of the log and ability to retrieve recorded log entries as desired. This functionality is encapsulated within the Logger class. And all logging entries are persisted in a single log file within the repository. Figure 8.12 depicts the Logger class and the LoggerLevel enumeration.

Figure 8.12: Logging Subsystem Class Diagram

8.4 Differential Calculation

A differential is the difference between two files and its calculation is one of the most important capabilities of a VCS. It supports the differential storage scheme, comparison

between artifact versions and the automatic merging of two branches. While branching and merging capabilities are not planned to be included in this version, the basic differential calculation will be included to provide a bases for following work.

8.4.1 Abstraction Layer

In this section an abstraction of the differential's change set will be designed to provide an abstraction between the calculation and representation of the differences between specified files. This decoupling is important because there is a wide variety of differential calculation algorithms each exhibiting assorted strengths and weaknesses. Some are appropriate for text based documents; while others are appropriate for specialized formats, such as XML. This separation will allow any differential algorithm to be employed within this system without needing to alter the system and simplifies its adjustment to operate on different types of artifacts.

Differential Interface

The differential is set of changes that must be made to artifact to transform it into another. The two artifacts may either be different artifacts or different versions of the same artifact. Therefore, the basic behavior of the differential interface takes two artifacts and returns the set of changes. Figure 8.13 depicts the resulting Differential interface.

Figure 8.13: Differential Interface

Change Set

A change set describes the alterations that must be made to an artifact to transform it into a second artifact. As described in [HM76] and [Mye86], the changes needed to transform a text document can be reduced to a set of two changes: (1) adding lines and (2) deleting lines. This can be observed in a simple, though inefficient example. First, delete all lines from the first file and then add all the desired lines from the target file. A third type, (3) replacing lines combines both operations when operating on the same set of lines and is commonly used. These three operations represent the set that must be recognized by this system.

In order to describe the different types of operations, either a conditional type field or a inheritance hierarchy may be used. In order to support possible other change operations the inheritance hierarchy option was chosen. Figure 8.14 on the following page depicts the resulting ChangeSet and Change hierarchy class diagram. The ChangeSet

consists of an arbitrary number of changes, which may be any of the following classes: ChangeAdd, ChangeDelete or ChangeReplace.

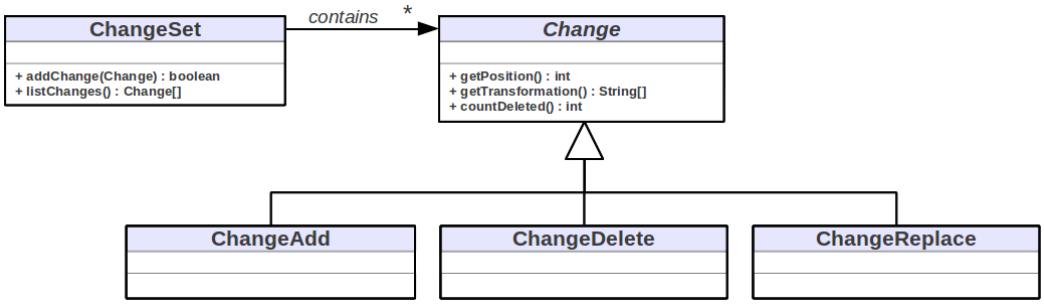


Figure 8.14: ChangeSet and Change Class Diagram

8.5 Summary

In this chapter, the design of the User Interface (UI) enabling interaction with a user and the various subsystems supporting the versioning core were introduced. First, the underlying command pattern and regular expression parsing concepts were described. Then the overlying UIs, CLI and GUI, which provides the user access to the functionality encapsulated within the commands, were designed.

Next, the various subsystems, i.e. persistency, logging and differential calculation, were described. The persistency subsystem employs the DAOs pattern to decouple the system from the underlying persistency infrastructure and the strategy pattern to allow the system to be configured to the user's needs. The logging component provides a lightweight logging implementation tailored to the needs of the system. Finally, the interface describing how a differential is calculated and represented within the system was developed. This will allow the system to be extended to implement an arbitrarily set of differential algorithms.

9 Design – Alpha-Flow Integration

In this chapter, an interface supporting α-Flow's versioning needs will be designed. The interface is VCS agnostic and capable of being implemented using any suitable VCS.

9.1 Roles

A VCS observes and records the history of an artifact much like a historian observes and records the history of mankind. It records information about the various states of the artifact during its evolution and provides access to this persisted information. Just as a historian writes their accounts in history books that may be read by others. The abstractions of this chapter will draw from this realization.

9.1.1 Historian Role Definitions

When considering the roles played by a historian, it may be observed that they play two primary roles: (1) investigative and (2) manipulative.

Manipulative The manipulative role alters the record of history either by appending new information or altering the perception of previously recorded information. A journalist recording the current events is an example of of an actor that appends new observations to history. An archaeologist excavating an ancient grave site of a little known culture is an example of an actor that alters the previously recorded perception of history, based on new observations.

Investigative The investigative role is responsible for analyzing and providing access to historical information for a given purpose. An actor that plays this role is a history professor at the university or a stock market analyst that analyzes the past to decipher trends that may be applied in the future.

Account The history on which each role operates and exchanges may be abstractly considered a historical account. These historical accounts represent a specific incidence or occurrence that may be either individual or within its evolutionary context considered.

While a single actor often plays both roles, a clear distinction based on the purpose of the task, either investigative or manipulative, may be identified. Figure 9.1 on the next page depicts the relationships between the Historian, Investigative Historian, Manipulative Historian and Historical Account roles.

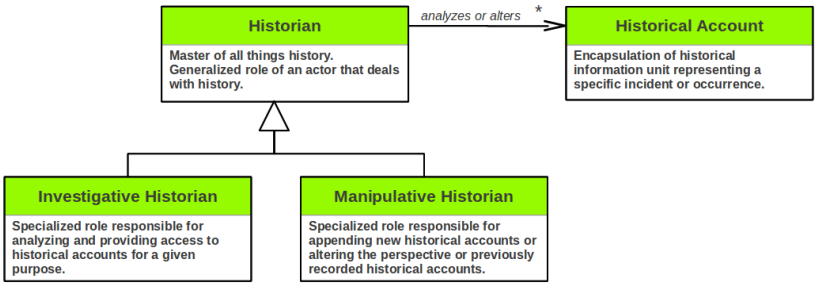


Figure 9.1: Historian Roles

9.1.2 Mapping Historian Roles to Version Control Systems

The abstraction of historical support within an VCS may be mapped by the previously defined historian roles. The composition of the investigative and manipulative roles provide the unification of the versioning responsibilities. A version may be abstractly considered a recorded account of the artifact's history. Figure 9.2 depicts the mapping the VCSs responsibilities on the previously described historian roles.

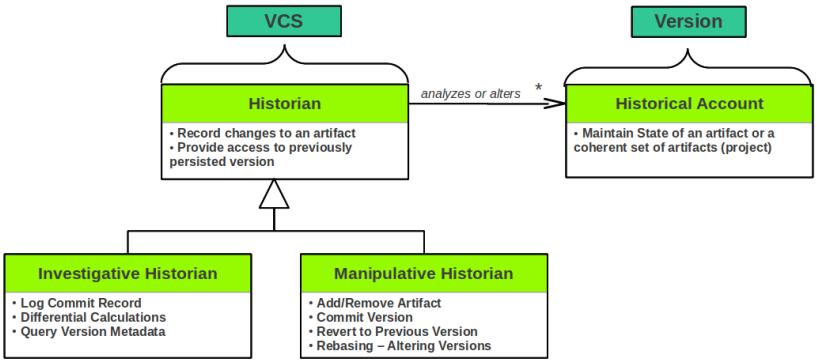


Figure 9.2: VCS Responsibilities Mapped to Historian Roles

Through these abstractions a VCS agnostic interface will be developed. The interface will provide a decoupling of the α-Flow project from the underlying VCS employed to support the versioning needs defined by this interface.

9.2 Alpha-Flow Interface Design

In the previous section, the abstract roles to be employed within α-Flow was presented. In this section, the functionality associated with these roles will be derived from the versioning needs of α-Flow and a concrete interface will be designed.

9.2.1 Mapping Alpha-Flow Requirements to Abstract Roles

The versioning needs of α-Flow have been introduced in Chapter 4: *Analysis – Alpha-Flow* on page 13 and will be summarized here. The following is a list of the defined needs and the associated number defines which role is responsible for providing the capability: (1) inspective, (2) manipulative or (3) account.

- Inspective
  - describe an α-Card evolution
  - traverse the evolution of an α-Card either along the system or valid path
- Manipulative
  - record changes to α-Cards and party responsible for making the changes
  - enable the restoration of α-Cards to previously persisted versions
  - create and identify designated versions as temporary
  - replace or update persisted version's content or metadata
  - insert a new version before a designated version or equivalently reorder versions
- Account
  - support the querying of α-Card version metadata
  - differentiate between valid and invalid states

9.2.2 Interface Definition

Combining these needs with the intimate knowledge of the α-Flow system, the following set of interfaces may be defined in Figure 9.3 on the next page. There are two noticeable structural changes: (1) Historical Account has been renamed to Version and (2) Historian is an abstract class that implements both the HistoryInvestigator and HistoryManipulator interfaces. The first improves the comprehensibility of the interface. The second is because of the Java programming semantics, specifically an interface cannot implement another interface. An abstract class can be used for that purpose.

This interface marries well with the designed Hydra VCS but also provides a layer of abstraction that frees an implementation to employ an arbitrary VCS.

9.3 Summary

In this chapter the abstract interface that provides a decoupling between the α-Flow project and the underlying VCS was designed. First, the abstract roles of a human historian and how they correspond to the responsibilities of a VCS was considered. Finally, based on the example and predefined versioning requirements, an abstract versioning interface for α-Flow was defined.

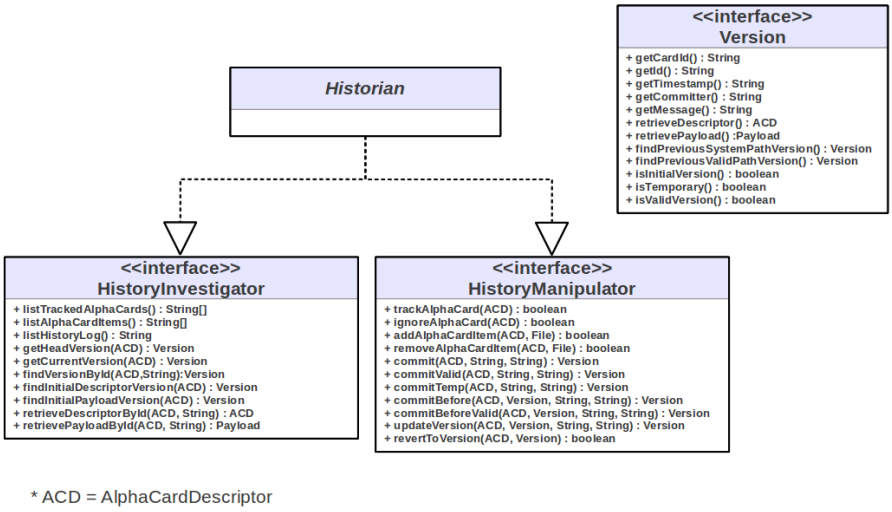


Figure 9.3: Alpha-Flow Abstract Versioning Interface

10 Implementation

In this chapter the implementation of the described design will be presented. Instead of presenting a rote recital of the actions taken, this chapter will focus on the issues that were found to challenge the system's implementation. The implementation follows a general path similar to that described during the system's design. First, the versioning core was implemented. Next, the user interfaces and subsystems were implemented. Finally, the α-Flow interface was defined, implemented and utilized.

10.1 An Agile Approach

Agile software development is a group of software development methodologies based on iterative and incremental development, where requirements and solutions evolve through cooperation between a team of developers, customers and users [Lar04].

No specific agile method was strictly followed, rather a custom mix of various methods was used. The mix was heavily influenced by eXtreme Programming (XP) [Bec99] and Scrum [Sch95] for the overall iteration planning, Kanban [Ras10] for the weekly execution efforts and Test Driven Development (TDD) [Kos08] at the lowest level of daily implementation.

10.1.1 Iterations

Development at the highest level preceded through four one month iterations. The first two iterations focused on development of the versioning core. The third focused on developing the supporting UIs and subsystems. The fourth dealt with the system's integration into α-Flow.

While each iteration had an identified focus, nothing prevented work outside of this focus from being included. Prior to each iteration, a planning meeting was held where the goals were solidified and prioritized. The focus of these planning meetings was to identify and organize the larger more abstract goals. No significant effort was wasted attempting to analyze the effort for each goal. This made it impossible to estimate what would be done in the iteration, but helped guide the developmental effort.

10.1.2 Task Definition and Execution

The goals were then broken into separate subtasks, which were also prioritized. Work progressed throughout the weeks of an iteration in a method described by Kanban. The

first task in the prioritized list was taken, implemented and then tested. By moving the tasks across the visualized board it was clear to see how work was progressing.

This allowed the most important work to be done and reduced the developmental overhead, by not attempting to force work to be compressed into weekly cycles as is expected in Scrum. Weekly meetings were then used to review the developmental progress, prioritize the backlog and refine any goals. This ensured that the product continued to evolve in response the changing demands of the client.

10.1.3 Assessment

This type of agile development may not be acceptable for industrial software production where there is a great need to develop a clear work schedule for the developers and plan future development around the product's progression. But showed itself to be extremely flexible and responsive to the changes uncovered during the development of a new conceptual product where uncertainty and false expectations are the norm. It provided the ability to guide development into uncertain areas through manipulation of the overarching concerns.

10.2 Versioning Core

The implementation of the versioning core was relatively clearly defined by the design and was realized in two steps. First, the retrievable elements and then, the committable elements were implemented. The majority of the implementation was relatively simple and requires no explanation. However, two area's were not sufficiently covered in the design and require some clarification: (1) calculation of the fingerprint's UID and (2) management of system configuration data.

10.2.1 Fingerprint Calculation

The SHA-1 algorithm take a series of bytes and creates a 160-bit hash that uniquely identifies the content. In order to improve the human readability of the fingerprint, the 160-bit hash is transformed into and maintained as a 40-digit hexadecimal hash. The fingerprint must be capable of creating the UID based on either an artifact's represented file or a container or state's metadata, which is dynamically produced.

The class java.security.MessageDigest provides the necessary capabilities for calculating a unique hash based on a given content. The steps for calculating the fingerprint's hash include: (1) get instance of SHA-1 message digest, (2) compute designated content's byte representation, (3) update message digest with the content's byte representation and (4) calculate the SHA-1 hash. This generalized algorithm is applies to each of the retrievable elements, i.e. artifact, container and state.

Artifact Fingerprint Calculation

The artifact's content bytes may be directly retrieved from the represented workspace file and feed into the message digest. Figure 10.1 depicts the artifact specific algorithm.

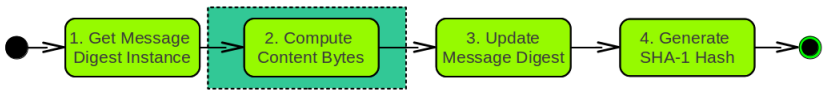


Figure 10.1: Calculation of Artifact Fingerprint

Container Fingerprint Calculation

Calculation of a container's fingerprint follows the same general algorithm. However, the second step must be expanded to encapsulate each contained element, i.e. artifact or container, and their fingerprint. This creates a recursive operation for the calculation of the subcontainers' fingerprint. Figure 10.2 is a depiction of the calculation of a container's fingerprint. The dark green hashed box highlights the portion of the algorithm that alters from that used in the artifact's fingerprint calculation.

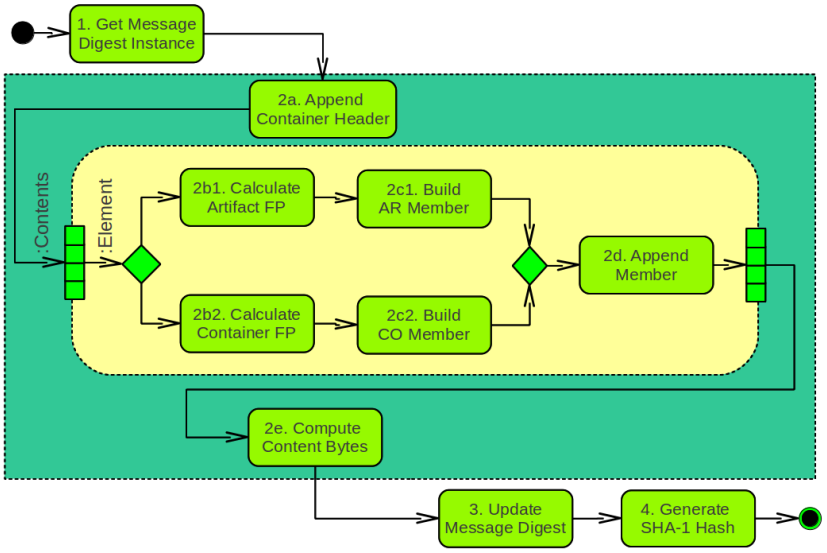


Figure 10.2: Calculation of Container Fingerprint

State Fingerprint Calculation

The calculation of the state's fingerprint also follows the same general algorithm and is simplified by the removal of the contents, metadata and previous states from the fingerprint calculation. Only the state's UUID member is required to be included. An activity diagram that depicts the calculation of the State's Fingerprint is presented in Figure 10.3. The dark green hashed box highlights the altered area.

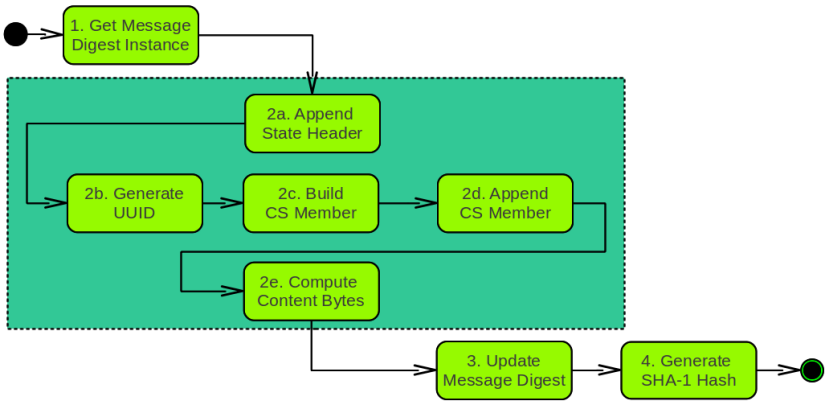


Figure 10.3: Calculation of State Fingerprint

10.2.2 Maintaining Configuration

One aspect of the program that was not covered in the design was the maintenance of the system's configuration. The configuration must be:

- commonly shared across the entire system
- notify system when it has been altered
- employ a simple and extensible interface
- persist the system's settings within the repository

Singleton Pattern vs. Static Access

In order to be commonly shared across the entire system it must either be a static or singleton class. The *singleton* design pattern "*ensure*(s) *a class only has one instance, and provide*(s) *a global point of access to it*" [GHJV95]. *Static* is an access specifier that enables a class member or method to be used independently of any object of that class [Sch02]. Based on these definitions, the singleton design pattern fits the intent of the Configuration class better and was thus chosen.

Observer Pattern

When the configuration has been altered, it must notify the appropriate portions of the system. This will allow them to adjust themselves to conform to the changes. In order to accomplish this the observer design pattern is applied. The observer design pattern "*define*(s) *a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically*" [GHJV95]. Thus, arbitrary extensions, such as the Logger or UI, may be created and listen for changes to the configuration without increasing the complexity of the system.

Name-Value Pairs

The configuration must provide an extremely flexible interface that allows it to support any number of arbitrary configuration details necessary. Assuming that any configuration detail can be described as a name-value pair, then it the configuration could provide an encapsulation of the Java class, Properties. However, a couple of special parameters, such as the workspace's and repository's address, are appropriate.

Property File Persistence

Of course the system's settings should not be lost when the system exits. Therefore, the configuration's details must be persisted within the repository and loaded when the system starts. Java's Properties class provides methods to support the storing and loading of the encapsulated properties in a designated file. The file, *hydra.properties*, will be reserved for this purpose.

Class Definition

Figure 10.4 is a depiction of the resulting Configuration class diagram. The Observable class and Observer interfaces are defined within the Java specification. The Configuration class inherits from the Observable class and thus inherits its capabilities. Any class wishing to be informed of any changes to the configuration, must implement the Observer interface and register itself as an observer of the configuration.

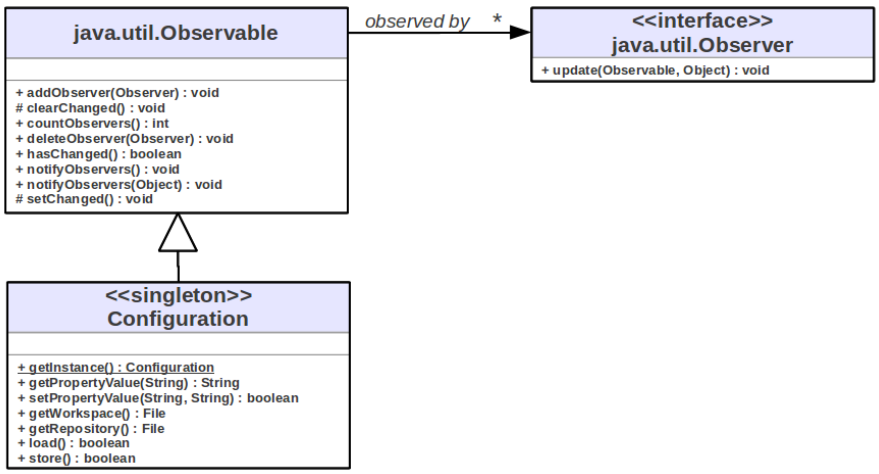


Figure 10.4: Configuration Class Diagram

10.3 User Interface

The majority of the UI's implementations were conducted without problem and in general not noteworthy. However, the actual regular expressions used for the commands

and the GUI's visualization implementations were of interest.

10.3.1 Command Regular Expressions

The key benefit of the RegExCommand is that the entirety of the command's definition and implementation is integrated into a single object. Any changes to a command may be carried out within a single decoupled class and have no impact of the rest of the system.

Command and Parameter Specification

The definition and parsing of a command line is reduced to the definition of a single regular expression, which is broken into two parts: (1) command and (2) parameters. The command portion determines if the command recognizes a given command line input, i.e. assumes that it is the intended command to be executed. The parameters enable the extraction of the command's specific parameters from the groups defined in the regular expression. Figure 10.5 is a depiction of a possible command line input that may be used to add a file, *testFile1.txt*, to the logical unit, *luX*.

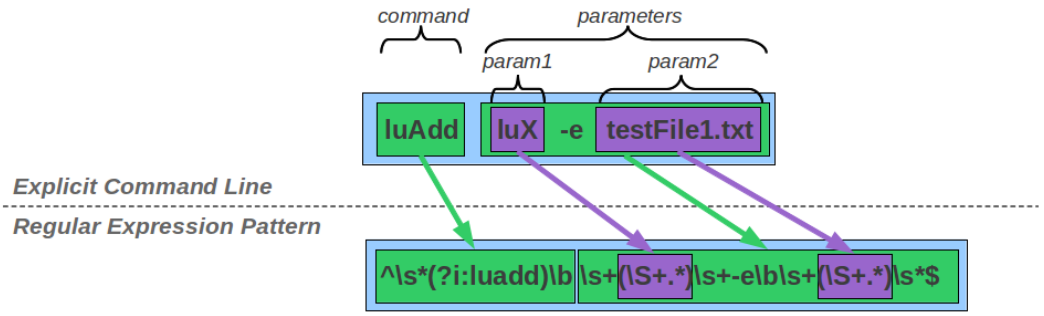


Figure 10.5: Example Regular Expression Pattern

Implementing New Commands

The ability to recognize a command line input and extract the necessary parameters through the use of regular expressions drastically simplifies the CLI command line parsing implementation. Additionally, this approach creates a truly pluggable interface for the introduction of new commands.

To implement a new Command class, the command line regular expression must be defined and the #getCommandRegex(), #extractParameters() and #execute() must be implemented.

The first method is a factory method that simply returns the defined regular expression pattern. The second method uses the groups defined in the regular expression to directly access the matched pattern. The third method is the heart of the command and realizes the represented behavior.

10.3.2 GUI Visualization

The majority of the GUI is implemented using Java Swing and presents no issues worthy of note. However, the visualization of a history as a graph employs the Java Universal Network/Graph Framework (JUNG)[1] to provide its depiction and manipulation.

JUNG Dependency

This is the only portion of the entire system that exhibits an external dependency beyond that of the standard Java 6 installation. Not only does it create an external dependency, it also dramatically increases its footprint. Whereas, Hydra's versioning core accounts for approximately 50.4 KBs, JUNGs dependencies account for 1.9 Megabytes (MBs). Clearly stated, it increases the footprint approximately 38.6 fold.

JUNG Benefits

However, the benefits provided by JUNG outweighed the negatives. JUNG supports the modeling, analysis and visualization of data that is represented as a graph. The capabilities relevant to this application are its ability to:

- define a network or graph as a set of nodes and arcs
- format a graph according to a given layout
- color and add text to nodes and arcs
- zoom and transpose the graph
- select and edit visible nodes or arcs

Screenshot

Figure 10.6 on the next page is a screenshot of the resulting GUI implementation and displays the graphical visualization provided through the JUNG framework. It depicts the history of the logical unit *yyy*, which has made a total of 13 commits and the second most recent state is current state represented in the workspace.

Because the GUI is not part of the core implementation, it must be decoupled to the point that it is not included in the core's JAR file. This will prevent the GUI's use of JUNG from disrupting α-Flow's lightweight ambitions.

10.4 Subsystems

The majority of the subsystems were developed to support the versioning core though abstraction and provide lightweight implementations for common tasks. For example, the persistency subsystem provides an decoupling of the system from the underlying

[1] JUNG Homepage: `http://jung.sourceforge.net`

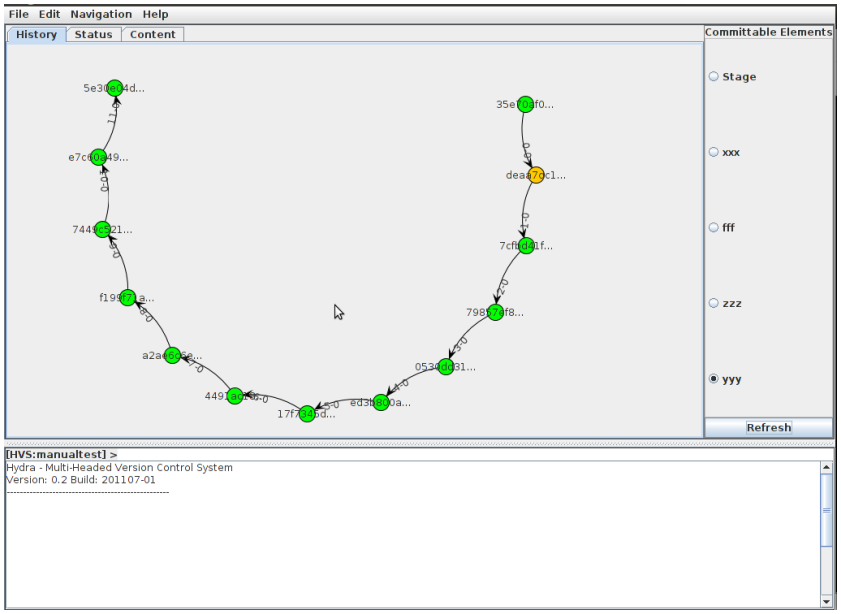


Figure 10.6: GUI Screenshot

persistence infrastructure and the Logging subsystem provides a lightweight implementation to avoid external dependencies and reduce its footprint. The implementation of both of these components requires no additional explanation beyond that covered in their design.

10.5 Alpha-Flow Integration

The implementation of the α-Flow interface also proceeded without note. However, the integration and the employment of the interface by the alpha-Properties component was achieved through the joint work between the author and implementer of the alpha-OffSync component.

10.6 Summary

This chapter covered some of the noteworthy details of the project's implementation. The vast majority of the implementation proceeded without any notable actions. However, there were several areas that were needed to be expanded or were otherwise interesting. Three areas, (1) fingerprint calculation, (2) configuration management and (3) regular expression based CLI processing, were of particular interest during the implementation. The GUI's use of the JUNG framework was also briefly mentioned.

11 Assessment

In this chapter an assessment of the project will be provided. First the achievement of the project's overall goals will be considered. Next, the project's software metrics and performance evaluation will be presented. Finally, areas for future work will be identified. Future work will include both improving the maturity of the Hydra mVCS and introducing new versioning concepts such as: (1) two-dimensional versioning, (2) hybrid repositories and (3) object versioning.

11.1 Overall Assessment

This project resulted in the design and successful implementation of Hydra, a mVCS providing explicit support for several new concepts. These new concepts include multi-headed versioning and validity tracking. Additionally, a VCS agnostic interface was developed for the α-Flow project and implemented employing the developed mVCS. During the project's development all of the defined requirements were successfully completed.

11.2 Software Metrics

Over 16,000 lines of code were produced during the development of the project. Executable code lines account for approximately 20 percent, test code accounts for approximately 28 percent and the remaining 52 percent is accounted for by blank lines and documentation. Figure 11.1 depicts the code distribution.

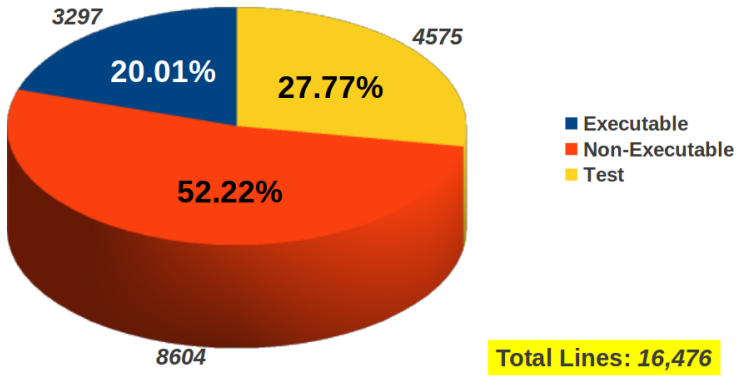


Figure 11.1: Lines of Code Distribution

The entire source code produces a 189.8KB JAR file. However, there is a significant amount of extra functionality, e.g. CLI and GUI, that is not necessary for supporting the α-Flow project. After stripping away unnecessary functionality, the resulting JAR file is 50.4 KBs. Figure 11.2 provides a visual presentation of this reduction. This is approximately 380 times smaller than the executable employed by Git, which is approximately 19.1 MB [Git11].

Figure 11.2: Core Executable Size

Testing was an important aspect of the developmental environment. Unit testing was implemented using the JUnit[1] framework [TLMG10]. Test coverage and a static analysis of the code was provided by the Cobertura[2] tool. A hand-rolled command line tool, JUnit Runner and Reporter (JuRR), provided a visual summary of the executed tests and identified any failures or errors. 285 JUnit tests took less than three seconds to execute and provided over 90 percent line coverage and over 85 percent branch coverage, including the GUI code. A McCabe's cyclomatic complexity rating of 1.933 was calculated. Figure 11.3 provides a graphical summary of the test coverage.

Figure 11.3: Test Coverage Summary

Line coverage refers to the number of lines that were executed at least once by the tests. Branch coverage refers to the number of paths or alternatives that were chosen for conditionally executing blocks of codes. McCabe's cyclomatic complexity provides a measure of the analyzed code's complexity based on its looping pattern. The complexity can be measured by counting the number of separate areas created by the code's control flow diagram. Alternatively, one may count the number of edges, subtract the nodes and add two to the result to attain the measure of complexity.

Coverage of at least 85 percent of the code is considered well tested [Kos08] and a McCabe's cyclomatic complexity of 10 or less is recommended [WM96]. Based on these metrics, the project's code is well tested and exhibits a low level of complexity. Finally, the test execution time of less than three seconds is very good and allows the test suite to be executed often.

[1] JUnit Homepage: `http://www.junit.org`
[2] Cobertura Homepage: `http://cobertura.sourceforge.net`

11.3 Functionality Evaluation

In this section an evaluation of Hydra's functionality is provided. The system supports multi-headed versioning and aspects of validity tracking which other VCSs do not explicitly support. Other benefits include its platform independence, embedability and light footprint. However, it does not support branching and inter-repository collaboration, two important capabilities associated with VCS technology.

Hydra supports all of the functionality intended to be implemented during the course of this project.

11.4 Performance Evaluation

Hydra's design and implementation emphasized the introduction of new functionality not performance optimization. However, to build a objective analysis of the system, its performance must be measured with respect to the performance of other VCSs. As described in Section 7.1: *Versioning Core* on page 67 the critical functionality of an VCS hinges on its ability to store and transfer data between the workspace and repository. Therefore, Hydra was tested to assess its ability in these key areas: (1) data transfer rate and (2) data compression.

Testing the data transfer rate was accomplished by measuring the time in which the system needed to execute the common versioning tasks of: (1) adding artifacts, (2) committing the state of the artifacts and (3) returning the artifacts to a previously persisted state. Testing the data compression was accomplished by measuring the resulting repository size after committing a large artifact set.

Hydra's performance in these categories depends heavily on the storage strategy employed. The differing storage strategies, i.e. NIOStorageStrategy, ZipStorageStrategy and GzipStorageStrategy, emphasize differing aspects of performance. The performance evaluation will provide a relative assessment of the different strategies.

11.4.1 Test Benchmark

While testing the various storage strategies will provide a relative measure of their performance, their performance should also be evaluated against mature VCSs that commonly employed. SVN's and Git's performance will be measured given the same tests in order to establish a benchmark against which Hydra's performance may be compared. SVN provides a benchmark representing the classical differential client-server architecture. Git represent the new dVCS and full copy storage.

11.4.2 Test Plan and Execution

The test will be executed in five steps; three of which were timed. In order to reduce the risk of human-error, the test steps were scripted and the timed tasks were evaluated using Linux's time command. The steps of the test plan are as follows:

1. *Initialize* new repository (not timed)
2. *Add* target artifact set
3. *Commit* workspace's state
4. *Delete* target artifact set (not timed)
5. *Revert* workspace to committed state (i.e. retrieve target artifact set)

The test data, i.e. the artifact set, consisted of 2,874 files (983.4 MB) of mixed binary and text documents, but was predominately text based documents.

All tests were executed five times and the fastest and slowest times were removed to reduce the effect of any outliers. Each run of the test was executed on the same system under a similar workload, i.e. no other user programs executing. The testing environment was an Ubuntu Linux version 10.4 with a Intel Quad 4 processor, 8 GB Random Access Memory (RAM) and a Western Digital 500 GB Blue Edition hard drive.

11.4.3 Test Results

Table 11.1 depicts the test results. Git's and SVN's test results are indicated on the left and provide a benchmark against which Hydra's various configurations are compared. Each of Hydra's three tested configurations is represented with two columns of results. The first column represents the actual measured time of execution. The second column provides a factor relating the configuration's performance to the two benchmarks. The first factor is with respect to Git's performance and the second factor is with respect to SVN's performance.

For an example, the overall time performance measure will be discussed. In order to accomplish all three timed tasks, Git required a total of 30.279 seconds and SVN took a total of 101.410 seconds. Hydra's NIO configuration required a total of 65.161 seconds. Compared to the benchmarks, it was 2.2 times slower than Git but 40 percent faster than SVN. The other configurations may be similarly understood.

Task	Git	SVN	NIO		Zip Compress		Gzip Compress	
Add	17.749s	7.465s	30.343s	x1.7 / 4.1	49.669s	x2.8 / 6.7	49.399s	x2.8 / 6.6
Commit	5.372s	56.881s	26.859s	x5.0 / 0.5	26.857s	x5.0 / 0.5	26.936s	x5.0 / 0.5
Retrieve	7.158s	37.063s	7.960s	x1.7 / 0.3	11.812s	x1.7 / 0.3	11.497s	x1.6 / 0.3
Overall	30.279s	101.410s	65.161s	x2.2 / 0.6	88.338s	x2.9 / 0.9	87.832	x2.9 / 0.9
Size	187.1MB	201.1MB	844.1MB	x4.5 / 4.2	173.2MB	x0.9 / 0.9	172.8	x0.9 / 0.9

Table 11.1: Stress Test Results

11.4.4 Assessment

All Hydra configurations had no problem managing the large data set. With respect to the data transfer rate, each of the configurations responded slower than Git but faster

than SVN. The NIO configuration was approximately 2.2 times slower than Git and was approximately 40 percent faster than SVN. The compressed configurations were approximately 2.9 times slower than Git and were about 10 percent faster than SVN.

Both of the compression strategies produced repositories that were approximately 10 percent smaller than either SVN or Git. However, the NIO configuration's repository was between four and five times larger than the other repositories.

While the system performed well in this evaluation, its performance could be improved. Several performance optimization techniques, which are introduced in Section 11.5.1: *Maturity Work* on page 125, may be employed to improve the system in these and other areas.

11.5 Future Work

In this section opportunities for future work are identified. The future work is broken into two categories: (1) maturity work and (2) conceptual work. Hydra mVCS provides an initial basis for building a fully functional and innovative VCS. Depending on the interest of the developer, there is work available in many areas.

11.5.1 Maturity Work

Maturity work deals with bringing the system to a level comparable to the state of art VCSs in well developed and supported functional areas. This work requires the ability to critically consider the various applicable technologies or related works. Additionally, an innovator may develop a new solution that can be compared against a wide variety of functional implementations. Areas that may be of interest include:

- Branching, Merging and Differential Calculation
- Data Exchange, Replicated Data Consistency and Distributed Architectures
- Data Compression Techniques
- High-Performance Java (especially I/O)
- Developmental Environment Integration

11.5.2 Conceptual Work

Conceptual work deals with the introduction of new innovative ideas that have no current comparable implementation and has little or no support within conventional VCSs. This work requires creative thinking to solve problems that others have avoided. The following is a brief description of three possibilities.

Two Dimensional Versioning

Rebasing and other techniques have introduced the ability to change a previously committed state. However, each change to a state should be somehow maintained and

possibly reinstated dependent upon the situations need. This introduces the complex subject of managing the versioning of a version and the ability to propagate changes forward through an artifact's history from a changed version into the current workspace artifact.

Hybrid Repository

Maintaining a complete copy of all artifact versions is the current approach to supporting distributed version control. However, this creates an enormous storage space requirement which consists mainly of information that will never be accessed. A hybrid repository attempts to combine the client-server's referential and the distributed full copy approaches into a intelligent repository that maintains an optimal amount of information locally while still providing the restoration of any previous state through referential means. This combines the benefits of both approaches with little negative impact on the system's functionality.

Live Object Versioning – Memento

Electronic documents are not the only electronic form that experiences an evolution that is worth recording. However, contemporary VCSs are designed specifically to track changes to electronic documents. Objects within a live system experience an evolutionary process and the recording of their evolution may be beneficial.

One such example is the state of a virtual reality training simulator, e.g. driver's training. The state of the simulation is comprised of a complex relationship between numerous objects, each maintaining their own state. The Memento pattern, extended to support an arbitrary object, may provide a basis for recording and restoring an object's state and thus the capability to generally persist and restore any live system's state.

11.6 Summary

This chapter provided an assessment of the project's overall success. An analysis the source code distribution and performance of the Hydra mVCS was conducted. Hydra is a well tested, lightweight embeddable mVCS that provides support for multi-headed versioning and validity tracking not found in other VCSs. However, it lacks in the maturity of other VCSs and does not support several common features associated with a VCS, namely branching and inter-repository data exchange.

Several opportunities for further work, both maturity improvements and VCS innovations, were introduced. Work improving the maturity of the system requires the ability to critically consider the various applicable technologies or related works and would result in the production of a fully functional system that may be employed in the real world. Conceptual work extends the system in areas that have either not been considered or have found little support by contemporary VCSs. These innovative ideas include: (1) two-dimensional versioning, (2) hybrid repositories and (3) live object versioning.

12 Conclusion

Version Control Systems (VCSs) have benefited from decades of continued evolution and development. Today they are an essential part of any software developmental effort and are used in other fields where the tracking of changes to electronic documents may be legally required or otherwise beneficial. However, they are lacking in two areas: (1) support for independent management of subprojects and (2) differentiation between versions based on their validity characteristics.

The restrictiveness of the first shortcoming forces an inefficient lockstep approach of development where the progress of each is limited to the speed of the slowest. This occurs because only a single monolithic state for the overall project's state is recorded which maintains all artifact interdependencies. Separating the overall project into independent logical units and allowing each to evolve autonomously frees development from the unnecessary dogmatic developmental style where all subprojects must progress in unison. Each logical unit maintains the intra-unit artifact dependencies while an overall project state reflects the integration of the various subproject states and maintains the inter-unit artifact dependencies. This improves the system testing and integration reflects the optimal combination of subproject states, instead of the simply recording the most recent states.

The inability to derive the validity of a state either forces the repetitive assessment of the state each time it is considered or an external support system must be developed to manage the information. Currently state of the art systems employ a propose-filter-accept workflow that eliminates unacceptable states from consideration. In this workflow all commits are initially proposed to a set of intermediary repositories. Next, valid states are filtered, either manually or automatically based on a suite of tests, from the intermediary repositories and transferred into a blessed repository. The blessed repository reflects the current acceptable state of the project. Developers then pull from the blessed repository and the cycle continues. However, these systems exhibit several handicaps:

- require significant overhead in time, space and man-power
- introduce a delay in the developmental cycle
- fail to provide a complete evolutionary account

By introducing the capability of defining the validity of a state to the VCS, these handicaps are reduced. The definition of validity may be based on the internal characteristics of the state, i.e. fails to compile or fails a given suite of tests, or its evolutionary context, i.e. from which state does it logically proceed.

This project designed and implemented a mVCS, Hydra, which supports these innovative concepts. It introduces the logical unit as a new versioning level of granularity and maintains validity as a property of a given state as well as a definition of its alternative history. Logical units allow for the independent evolution of subprojects while still maintaining the ability to reproduce any overall coherent project state. This improves the efficiency of the overall project development by reducing the restrictions placed on subprojects. The explicit maintenance of each state's validity and the valid evolutionary path reduces the overhead of its management and provides an more complete depiction of an evolution.

Appendices

A Hydra – Quick Start

This appendix provides a brief introduction to version control using Hydra – Multi-Headed Version Control System. It is assumed that the reader is familiar with multi-headed versioning. Very simply stated: each subprojects maintains its own head which and allows for their independent management. The overall project is managed by an element named the stage, which is responsible for maintaining the history of the project by persisting coherent states that encapsulate all of the project's subproject (i.e. logical units) states.

A.1 Installation

This section presents the basic requirements for setting up Hydra. Currently, there is no automatic installation process, but the system is very small and requires very little effort for setup.

The following is a listing of items are either required or optional:

- *hydra-0.2.jar* – Required. This is the entirety of the system.
- *hydra* (script) – Optional. Provides a simplified startup routine that identifies the classpath and may be used as the system's executable.
- GUI dependencies (i.e. JUNG) – Optional. Provides support for the GUI
  - *collections-generic-4.01.jar*
  - *colt-1.2.0.jar*
  - *concurrent-1.3.4.jar*
  - *jung-algorithms-2.0.1.jar*
  - *jung-api-2.0.1.jar*
  - *jung-graph-impl-2.0.1.jar*
  - *jung-visualization-2.0.1.jar*

The only item necessary to employ the system is the *hydra-0.2.jar*. The remaining items are optional. The JUNG dependencies may be downloaded directly from the JUNG Homepage[1].

[1] JUNG Homepage: `http://jung.sourceforge.net`

A.1.1 Organization

The general organizational structure is very simple. Place the *hydra-0.2.jar* file in a directory with the *hydra* script file and the dependencies in a subdirectory named *lib*. Alternatively, the dependencies may be directly added to the system's Java classpath.

A.1.2 Shell Script

The *hydra* shell script is a simple shell script that encapsulates the command line that executes the Hydra class in the JAR file. Listing: A.1 presents the contents of the shell script. As seen, it assumes all dependency JAR files are located in a subdirectory name *lib*. A similar script could be produced for any Operating System (OS).

```
#!/bin/bash

HYDRA_CLASSPATH=$PWD:$PWD/hydra-0.2.jar

for i in $PWD/lib/*.jar;
do
        HYDRA_CLASSPATH=${HYDRA_CLASSPATH}:"$i";
done

java -cp "${HYDRA_CLASSPATH}" org.hydra.Hydra $@
```

Listing A.1: Example Hydra Bash Script

A.2 Starting Hydra

Hydra may be started in three ways: (1) direct class execution org.hydra.Hydra, (2) executable JAR (*hydra-0.2.jar* or (3) script execution (*hydra* script). Described as follows:

1. java -cp *classpath* org.hydra.Hydra *parameters*
2. java -cp *classpath* -jar hydra-0.2.jar *parameters*
3. ./hydra *parameters*

A.2.1 Execution Modes

Hydra supports three execution modes: (1) single command, (2) interactive CLI and (3) interactive GUI. The interactive execution mode is derived from the given command line parameters; --cli and --gui requests execution in the respective interactive modes (e.g. ./hydra --cli will start Hydra in the interactive CLI mode). In the interactive CLI

mode, the system will continue to prompt for the next command until the exit command is entered. The interactive GUI mode exits when the application's window is closed or the *Exit* command is selected from the appropriate *File* menu.

Hydra is executed in single command mode when a command (e.g. help) is included in the parameters and no other mode specifier (i.e. either --cli or --gui is included). This mode will execute the given command and exit.

A.2.2 Initializing A New Repository

A repository is the location where the persisted versions and system management information is maintained. The repository is a folder in the root of the workspace that is named *.hydra*. When Hydra is started, it will search in the given workspace for a repository. If no repository is found, it will search recursively to the root of the file system. If no repository is found during this search the system will exit.

In order to initialize a new repository, the --initialize parameter is used. To initialize a new repository in the current working directory use the following command:

```
./hydra --initialize
```

A.2.3 Other Parameters

Here is a listing of other parameters that may be included to configure the system.

--cwd *path*	set current working directory to path
--v *0-10*	set system verbosity

To initialize a repository in the subdirectory *proj* and start an interactive CLI with a verbosity of 5, use the following command.

```
./hydra --cwd proj --initialize --cli --v 5
```

A.3 Creating and Managing Logical Units

Logical units provide a means for independently managing the evolution of subprojects. However, overall project is represented as a stage. The stage is capable of creating, managing and ignoring logical units. Additionally, to reduce the amount of typing required a specific logical unit may be the stage's focus. This assumes that all logical unit commands, which have no specified logical unit, will be executed with respect to the focused logical unit. The following is a listing of the specific commands that accomplish the previously described tasks:

sCreate *luName*	creates a logical unit with the designated name
sIgnore *luName*	ignore logical unit (i.e. do not include in stage commits)
sManage *luName*	starts managing a logical unit that is being ignored
sFocus *luName*	focus on designate logical unit

A.3.1 Stage and Logical Units

All commands that are executed under the authority of the stage are prefixed with the letter *s* and all that operate on a logical unit begin with the letters *lu*. The current status of the stage and a designated logical unit may be found through the following commands:

sStatus	display status of the stage
luStatus *luName*	display status of the designated logical unit

A.4 Dealing with Files

Arbitrary files may be added to either the stage or a logical unit. The location of the file is irrelevant to which element (i.e. stage or logical unit) it is added and a file may be added to more than one.

A.4.1 Listing Directory/File Contents

While in interactive mode, the current workspace's files may be queried using the list command, similar to either ls or dir. The default setting of the command is to only show the current level of the workspace. However, that may be adjusted by increasing the depth. Additionally, if a file is specified, the command displays the content of the designated file. The file is summarize as:

list {-d*depth*} {*path*}	display contents of directory or file

A.4.2 Adding and Removing Files

Files for which changes are to be tracked must be added to an element that is responsible for persisting a designated version and returning the file to a previously persisted version. Files may either be added to or removed from the stage or designated logical unit with the following commands:

luAdd *luName* -e{r} *file*	add file to designated logical unit tracking
luRemove *luName* -e *file*	remove file from designated logical unit tracking
sAdd -e{r} *file*	add file to stage tracking
sRemove -e *file*	remove file from stage tracking

A.4.3 File Differentials

The difference between the current workspace file and the last persisted or most recently reverted to version (i.e. the current version) may be queried through the diff command. The variants are as follows:

luDiff *luName* -e *file*	display diff between workspace and current version
sDiff -e *file*	display diff between workspace and current version

A.5 History Management

The stage and logical units are responsible for independently (1) tracking the changes to a designated set of files, (2) restoring the workspace to a previous version and (3) describing the element's evolution or history. However, the user must explicitly request these functions. The commit command records the current state of the element's workspace. The revert command returns the workspace to a previously persisted state. The log command provides a listing of the committed versions and their metadata.

A.5.1 Committing a Version

A commit is the act of persisting a specified version of a set of files. However, the stage is responsible for maintaining the state of not only a given set of files but also the overall project decomposed into the managed logical units. Therefore, the stage's commit encompasses the state both the set of files and all currently managed logical units. However, if logical units have been changed since their last commit, the stage has the ability to request that they all commit their changes before the overall project's state is recorded. This is known as a recursive commit. The following is a listing of the commit commands:

luCommit *luName* -m *message*	record logical unit's changes
sCommit {-full} -m *message*	record stage's and logical unit's changes*

The -full parameter causes the stage to request that each logical unit commit its changes before the overall project's state is committed.

A.5.2 Reverting the Workspace

Reverting is the act of return the workspace to a previous state. The previous state may be defined in three ways: (1) explicitly through the commit's fingerprint hash, (2) along a designated path from the element's head (i.e. most recently committed state) or (3) relative to the current state (i.e. the state that was either most recently committed or reverted to). The following is the list of commands that may be used to revert either a stage or a logical unit:

luRevert *luName* -h *fpHash*	revert to state with designated fingerprint
luRevert *luName* -p *path*	revert to state along designated path from head
luRevert *luName* -r *branch distance*	revert to state relative to current
sRevert -h *fpHash*	revert to state with designated fingerprint
sRevert -p *path*	revert to state along designated path from head
sRevert -r *branch distance*	revert to state relative to current

When defining a relative state, the branch is the position of the previous state in the current state's list and the distance is the number of previous states to move. The relative revert can move both forward and backwards along a history. The path is a

sequence of steps (i.e. branch and distance combinations) that may be traced to find the designated state. An example of a path is *1+2*2+1. The example path may be translated to say, take the first previous state and move to its previous and then take the second previous state from that state.

An additional command, reset, allows the designated element to restore the current or head persisted state of the workspace. This is very useful when one would like to restore the workspace to a consistent state after a failed developmental check. It is equivalent to reverting to the current or head state's hash or moving nowhere in a relative revert. This command is summarized as follows:

luReset *luName* {-C | -H}	reset logical unit's workspace to head or current state.
sReset {-C | -H}	reset stage's workspace to head or current state.

A.5.3 Logging a History

The actual evolution of an element may also be described through the log command. This command presents each commit and its metadata (i.e. responsible party, purpose, commit fingerprint hash and creational timestamp). The following is a listing of the log commands:

luLog *luName* {-S | -V}	display log of designated logical unit's commits
sLog {-S | -V}	display log of stage's commits

A.6 System Configuration and Commands

There are a couple of other commands that may be used to either configure or query information about the current status of the system. They include:

exit | quit	exit the command line interface
help	display help menu and list of command usage
log *num*	display the last *num* entries in system log
setUser *userId*	set the system's responsible party
status	display the system's status
verbosity *0-10*	set system's verbosity level

Hydra – Multi-Headed Version Control System
Version: 0.2 Build: 201107-1

Usage:

java -cp *classpath* org.hydra.Hydra {*params*} {*cmd*}
java -cp *classpath* -jar hydra-0.2.jar {*params*} {*cmd*}

Example: java -jar hydra-0.2.jar --cwd temp/manualtest --v 5 help

Params:

--cli	CLI - interactive mode
--gui	GUI - interactive mode
--cwd *path*	set current working directory
--initialize	initialize new repository
--v *0-10*	set system verbosity level

Commands:

exit(e)|quit(q)	exits from the CLI
help(h)	print this help menu
status(s)	print the system's status
log {*num*}	print last *num* system log entries
list(ls) {-d*depth*} {*path*}	print directory or file content
setUser *userId*	set the system's user id
verbosity(v) *0-10*	set the system's verbosity level
luStatus(lus) *luName*	print status of logical unit
luAdd *luName* -e{r} *file*	add file to logical unit
luRemove *luName* -e *file*	remove file from logical unit
luStash *luName*	store current contents in repository
luCommit *luName* -m *message*	commit designated logical unit
luDiff *luName* -e *file*	print file differential (wrt current)
luLog *luName* {-S | -V}	print commit log along system or valid path
luReset *luName* {-C | -H}	revert to head or current version
luRevert *luName* -h *fpHash*	revert to version with designated fingerprint
luRevert *luName* -p *path*	revert along path from head
luRevert *luName* -r *branch distance*	revert relative to current
sCreate *luName*	create new logical unit with designated name
sIgnore *luName*	ignore designated logical unit
sManage *luName*	track designated logical unit
sFocus *luName*	focus on designated logical unit
sAdd -e{r} *file*	add file to stage
sRemove -e *file*	remove file from stage
sStash	store stage's content in repository
sStatus(ss)	print stage's status
sCommit {-full} -m *message*	commit stage (-full is recursive)
sDiff -e *file*	print file differential (wrt current)
sLog {-S | -V}	print commit log along system or valid path
sReset {-C | -H}	revert workspace to current or head version
sRevert {-full} -h *fpHash*	revert to version with designated fingerprint

Bibliography

[Bec99] Kent Beck. *eXtreme Programming eXplained*. Addison Wesley, 1999.

[BME+07] Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Yong, Jim Conallen, and Kelli A. Houston. *Object-Oriented Analysis and Design with Applications*. Addison–Wesley, 3rd edition, 2007.

[Bur95] James H. Burrows. Secure Hash Standard (FIBS PUB 180-1). Technical report, National Institute of Standards Technology, April 1995.

[Cha11] Scott Chacon, editor. *Git Community Book*. October 2011.

[CK03] William Crawford and Jonathan Kaplan. *J2EE Design Patterns*. O'Reilly, 2003.

[Coh10] Mike Cohn. *Succeeding with Agile – Softwware Development Using Scrum*. Addison Wesley, 2010.

[Fel79] Stuart I. Feldman. Make – A Program for Maintaining Computer Programs. *Software – Practice and Experience*, 9(3):255–265, March 1979.

[Fow99] Martin Fowler. *Refactoring – Improving the Design of Existing Code*. Addison Wesley, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[Git11] Git. Git Homepage. `http://git-scm.com`, October 2011.

[Gru86] Dick Grune. Concurrent Versions System – A Method for Independent Cooperation. 1986.

[Ham05] Hamill. *Unit Test Frameworks*. O'Reilly, 2005.

[HM76] J. W. Hunt and M. D. McIlroy. An Algorithm for Differential File Comparison. Technical report, Bell Laboratories, June 1976.

[HO11] Graydon Hoare and Others. *Monotone – A Distributed Version Control System*. 2011.

[HR83] Theo Haerder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM – Computing Surveys*, 15, December 1983.

[Hud11] Hudson. Hudson (Designing Pre-Tested Commitsr). `http://wiki.hudson-ci.org/display/HUDSON/Designing+pre-tested+commit`, October 2011.

[IEE90] A Glossary of Software Engineering Terminology. Technical Report IEEE 610.12, Institute of Electirical anbd Electronic Engineers, 1990.

[Jet11] JetBrains. Jet Brains Homepage. `http://www.jetbrains.com/teamcity`, October 2011.

[Kos08] Lasse Koskela. *Test Driven – Practical TDD and Acceptance TDD for Java Developers*. Manning, 2008.

[Lar04] Craig Larman. *Agile & Iterative Development – A Manager's Guide*. Addison Wesley, 2004.

[LED+99] Anthony LaMarca, W. Keith Edwards, Paul Dourish, John Lamping, Ian Smith, and Jim Thornton. Taking the Work out of Workflow: Mechanisms for Document–Centered Collaboration. In *Proc of the 6th European Conference on Computer–Supported Cooperative Work*, Copenhagen, Denmark, September 1999.

[LH07] Steve Loughran and Erik Hatcher. *ANT in Action*. Manning, 2007.

[Loe09] Jon Loeliger. *Version Control with Git*. O'Reilly, 2009.

[Mec05] Robert Mecklenburg. *Managing Projects with GNU Make*. O'Reilly, 2005.

[Mer11] Mercurial. Mercurial Homepage. `http://mercurial.selenic.com`, October 2011.

[MTM+07] J.D. Meier, Jason Taylor, Alex Mackman, Prashant Bansode, and Kevin Jones. *Team Development with Visual Studio Team Foundation Server*. Microsoft Corporation, 2007.

[Muk05] Patrick Mukherjee. *A Fully Decentralized, Peer-to-Peer Version Control System*. PhD thesis, 2005.

[Mye86] Eugene Myers. An O(ND) Difference Algorithm and its Variations. *Algorithmica*, (1):251–166, 1986.

[NL09] Christoph P. Neumann and Richard Lenz. alpha-Flow: A Document-based Approach to Inter-Institutional Process Support in Healthcare. In *Proc of the 3rd Int'l Workshop on Process-oriented Information Systems in Healthcare (ProHealth '09) in conjunction with the 7th Int'l Conf on Business Process Management (BPM'09)*, Ulm, Germany, September 2009.

[NL10] Christoph P. Neumann and Richard Lenz. The alpha-Flow Use-CAse of Breast Cancer Treatment – Modeling Inter- Instituational Healthcare Workflows by Active Documents. In *Proc of the 8th Int'l Workshop on Agent-based Computing for Enterprise Collaboration (ACEC) at the 19th Int'l Workshops on Enabling Technologies: Infrastructures for Collaaborative Enterprises (WETICE 2010)*, Larissa, Greece, June 2010.

[NL12] Christoph P. Neumann and Richard Lenz. The alpha-Flow Approach to Inter-Institutional Process Support in Healthcare. *International Journal of Knowledge-Based Organizations (IJKBO)*, 2, 2012. Accepted for publication.

[NSWL11] Christoph P. Neumann, Peter K. Schwab, Andreas M. Wahl, and Richard Lenz. alpha-Adaptive: Evolutionary Workflow Metadata in Distributed Document- Oriented Process Management. In *Proc of the 5th Int'l Workshop on Process-oriented Information Systems in Healthcare (ProHealth '11) in conjunction with the 9th Int'l Conf on Business Process Management (BPM'11)*, Clermont-Ferrand, France, July 2011.

[O'S09] Bryan O'Sullivan. *Mercurial: The Definitive Guide*. O'Reilly, 2009.

[PCSF08] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. Version Control with Subversion. 2008.

[Pea02] Judy Pearsall. *The Concise Oxford English Dictionary*. Oxford University Press, 10th edition, 2002.

[Ras10] Jonathan Rasmusson. *The Agile Samurai – How Agile Masters Deliver Great Software*. Programatic Bookshelf, 2010.

[Roc75] Marc J. Rochkind. The Source Code Control System. *IEEE Transactions on Softweare Engineering*, SE-1(4):364–370, December 1975.

[Sch95] Ken Schwaber. SCRUM Development Process. In *OOPSLA '95 Workshop on Business Object Design and Implementation*, Austin, Texas (USA), October 1995.

[Sch02] Herbert Schildt. *Java 2 – The Complete Reference*. McGraw–Hill, 5th edition, 2002.

[Sch05] Stephen R. Schach. *Object-Oriented & Classical Software Engineering*. McGraw–Hill, 6th edition, 2005.

[SS05] Yasuhi Saito and Marc Shapiro. Optimistic Replication. *ACM – Computing Surveys*, 37(1):42–81, March 2005.

[Tic85] Walter F. Tichy. RCS – A System for Version Control. *Software Practice and Experience*, 15(7):637–654, July 1985.

[TLMG10] Peter Tahchiev, Felipe Leme, Vincent Massol, and Gary Gregory. *JUnit in Action*. Manning, 2nd edition, 2010.

[TN11] Aneliya Todorova and Christoph P. Neumann. alpha-Props: A Rule-Based Approach to 'Active Properties' for Document-Oriented Process Support in Inter-Institutional Environments. In Ludger Porada, editor, *Lecture Notes in Informatics (LNI) Seminars 10 / Informatiktage 2011*. Gesellschaft fr Informatik, March 2011.

[vdAWG05] W. M. P. van der Aalst, M. Weske, and D. Gruenbauer. Case handling: a new paradigm for business process support. *Data & Knowledge Engineering*, 53(2):129–162, 2005.

[Wah11] Andreas M. Wahl. alpha-OffSync: Verteilten Datensynchronisation in Form von IMAP-basiertem Mail-Transfer als Baustein einer Prozessuntersttzung auf Basis von aktiven Dokumenten, December 2011.

[WM96] Arthur H. Watson and Thomas J. McCabe. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. Technical report, National Institute of Standards and Technology, September 1996.

[Wor11] Word Aligned. A Subversion Pre-Commit Hook. `http://wordaligned.org/articles/a-subversion-pre-commit-hook`, October 2011.